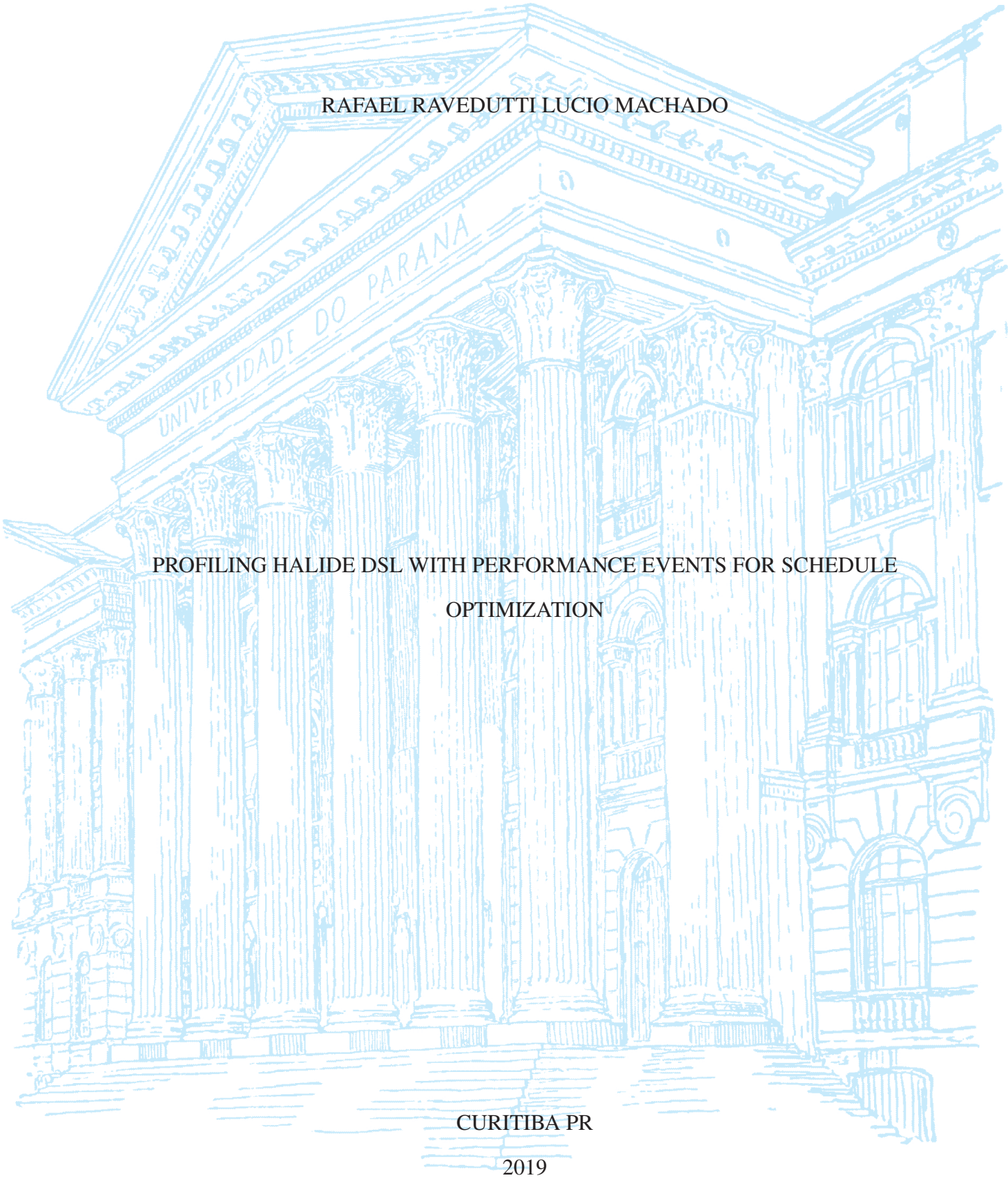
UNIVERSIDADE FEDERAL DO PARANÁ

RAFAEL RAVEDUTTI LUCIO MACHADO



PROFILING HALIDE DSL WITH PERFORMANCE EVENTS FOR SCHEDULE OPTIMIZATION

CURITIBA PR

2019

RAFAEL RAVEDUTTI LUCIO MACHADO

PROFILING HALIDE DSL WITH PERFORMANCE EVENTS FOR SCHEDULE OPTIMIZATION

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Daniel Weingaertner.

Coorientador: Andre Murbach Maidl.

CURITIBA PR

2019

Catalogação na Fonte: Sistema de Bibliotecas, UFPR
Biblioteca de Ciência e Tecnologia

M149p Machado, Rafael Ravedutti Lucio
Profiling Halide DSL with performance events for schedule optimization [recurso eletrônico] / Rafael Ravedutti Lucio Machado – Curitiba, 2019.

Dissertação - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-graduação em Informática.
Orientador: Daniel Weingaertner
Coorientador: Andre Murbach Maidl

1. Processamento de imagens. I. Universidade Federal do Paraná. II. Weingaertner, Daniel. III. Maidl, Andre Murbach. IV. Título.

CDD: 006.72

Bibliotecária: Roseny Rivelini Morciani CRB-9/1585

MINISTÉRIO DA EDUCAÇÃO
SETOR DE CIENCIAS EXATAS
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO INFORMÁTICA - 40001016034P5

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **RAFAEL RAVEDUTTI LUCIO MACHADO** intitulada: **Profiling Halide DSL with Performance Events for Schedule Optimization**, sob orientação do Prof. Dr. DANIEL WEINGAERTNER, que após após terem inquirido o aluno e realizado a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de mestre está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 01 de Agosto de 2019.

DANIEL WEINGAERTNER
Presidente da Banca Examinadora (UNIVERSIDADE FEDERAL DO PARANÁ)

ANDRE MURBACH MAIDL
Coorientador - Avaliador Externo (ELASTIC)

ANDRÉ RAUBER DU BOIS
Avaliador Externo (UNIVERSIDADE FEDERAL DE PELOTAS)

MARCO ANTONIO ZANATA ALVES
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

MINISTÉRIO DA EDUCAÇÃO - UNIVERSIDADE FEDERAL DO PARANÁ - Setor de Ciências Exatas - Coordenação de Pós-Graduação em Informática

“Science advances one funeral at a time.” - Max Planck

ACKNOWLEDGEMENTS

Primeiramente, agradeço a meus pais Vivian e Rafael por todo o apoio e suporte que me deram durante toda a vida, o que me permitiu ter a oportunidade de realizar o meu mestrado em uma universidade pública e prestigiada como a UFPR.

Agradeço a minha esposa Leticia que sempre esteve ao meu lado, e sempre me ajudou nos momentos de maior dificuldade.

Agradeço também aos meus orientadores, Daniel e Andre, que além de todo o suporte sempre me serviram como inspiração para entrar na vida acadêmica. Sem eles esse trabalho jamais seria possível.

Finalmente, agradeço aos participantes da banca examinadora responsável por este trabalho, que contribuiram com diversas ideias e sugestões, e cujas criticas foram muito importantes para o amadurecimento deste trabalho.

RESUMO

Aplicações de processamento de imagens atualmente requerem otimizações e especializações apropriadas para que possam atingir boa performance em hardwares paralelos e heterogêneos disponíveis. Um problema existente utilizando linguagens de propósito geral é que cada desenvolvedor precisa escrever diferentes versões da mesma aplicação para executar de maneira otimizada em diferentes tipos de hardware, o que requer mais trabalho e conhecimento destes desenvolvedores e também prejudica a manutenabilidade do software.

Halide é uma linguagem de domínio específico (DSL) para processamento de imagens que reforça a separação do algoritmo e do escalonamento de execução, permitindo a geração de código especializado para conjuntos de instruções de arquiteturas distintos reescrevendo apenas o escalonamento de execução, ao invés de todo o algoritmo. Halide torna muito mais prático escrever programas de processamento de imagens otimizados porque apenas o escalonamento precisa ser ajustado, no entanto escrever bons escalonamentos Halide não é uma tarefa fácil. Para alcançar bom escalonamento de pipelines Halide, ainda é necessário um profundo entendimento das arquiteturas alvo. Além disso, quando o tamanho e a complexidade do pipeline crescem, encontrar bons escalonamentos se torna muito mais desafiador para os desenvolvedores do escalonamento.

Para ajudar na criação de bons escalonamentos Halide, nosso trabalho estende a DSL Halide adicionando uma API de instrumentação que utiliza eventos de desempenho da CPU para medir eventos suportados pelo processador alvo durante a execução da aplicação. A extensão proposta oferece instrumentação de diferentes níveis de laço e relações de produção e consumo de funções, embutindo chamadas para uma biblioteca de instrumentação nos laços aninhados do código gerado. A extensão também suporta a instrumentação individual por threads em regiões paralelas. Como um caso de estudo utilizamos a biblioteca PAPI para contar eventos tais como misses na cache L1, número de operações em ponto flutuante (FLOP) e volume de dados da cache L3 em uma CPU Intel Core i5-7500, e discutimos como os resultados reportados podem ser utilizados para manualmente ou automaticamente gerar melhores escalonamentos para um pipeline de processamento de imagens.

Palavras-chave: processamento de imagens, profiler, linguagem de domínio específico

ABSTRACT

Image processing applications require proper optimization and specialization in order to achieve good performance in current parallel and heterogeneous hardware available. An existing issue with this approach on general purpose languages is that each developer needs to write different versions of the same application to execute optimally in different types of hardware, which requires more work and knowledge from application developers and also difficults software maintainability.

Halide is a domain-specific language (DSL) for image processing that enforces a separation of the algorithm and the execution schedule, allowing the generation of specialized code for distinct compute architectures by rewriting only the execution scheduler, instead of the whole algorithm. Halide turns much more practical to write optimized image processing programs because only the schedule has to be adjusted, thought writing good schedules for Halide is not an easy task. To achieve good scheduling of Halide pipelines, it is still necessary to have a deep understanding of the target architectures. Besides, when the size and complexity of the pipeline grows, finding good schedules becomes much more challenging for the schedule developers.

In order to support the creation of good Halide schedulers, our work extends the Halide DSL by adding a profiling API that uses the CPU Performance Events to measure events supported by the target processor during the application runtime. The proposed extension offers profiling of the application loop levels and functions' producer and consumer relations, embedding calls to a profiling library in the loop nests of the generated code. It also supports individualized profiling by threads on parallel regions. As a case study we use the PAPI library in order to count events such as L1 cache misses, number of float operations (FLOP) and L3 data volume on an Intel Core i5-7500 CPU, and discuss how the reported results can be used to manually or automatically generate better schedules for an image processing pipeline.

Keywords: image processing, profiler, domain specific language

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

AST	Abstract Syntax Tree
ARM	Advanced RISC Machine
AVX	Advanced Vector Extensions
BVH	Bounding Volume Hierarchy
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DAG	Directed Acyclic Graph
DSP	Digital Signal Processing
DSL	Domain-Specific Language
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
HIPAcc	Heterogeneous Image Processing Acceleration
HLS	High Level Synthesis
IR	Intermediate Representation
ISA	Instruction Set Architecture
LLVM	Low Level Virtual Machine
MPI	Message Passing Interface
MRI	Magnetic Resonance Imaging
NUMA	Non-Uniform Memory Access
OpenCL	Open Computing Language
OpenCV	Open Source Computer Vision Library
OpenMP	Open Multi-Processing
PAPI	Performance Application Programming Interface
PDE	Partial Differential Equation
SIMD	Single Instruction, Multiple Data
SSE	Streaming SIMD Extensions

CONTENTS

1 INTRODUCTION

Image processing algorithms are widely used in mobile and desktop computers for image edition/enhancement, and are frequently embedded in imaging equipment and digital cameras. The majority of these algorithms require good performance, specially on portable hardware such as mobile devices or cameras, demanding appropriate code optimization in order to run with a limited time and/or energy budget. Tuning image processing algorithms for optimal resource usage is not an easy task, because it requires a good understanding on how the code is executed in each target platform. Moreover, given the multitude of available processors and system configurations, finding the appropriate computation schedule to achieve the best performance on each of them can be a time consuming task, requiring multiple versions of the same code to be implemented and maintained.

For this purpose, we may use domain-specific languages for describing image processing algorithms in a more friendly way. Some approaches allow the developer to specify only the computation to be executed, and then the compiler is encharged for scheduling the computations and generating the final optimized version of the code for the desired architecture by using an auto-tuner. However, studies in these areas usually present some limitations, and some specific types of algorithms are not trivial to optimize automatically.

Another approach, which is more focused in this work, is the usage of a domain-specific language that allows the specification of the algorithm and its scheduling separately. This can help to split the work between the domain-specific language user, who just writes the high-level version of the code without worrying about how it will be executed in the hardware, and one or more schedulers, who specify how the code must be executed on one or more architectures to extract high performance.

Halide [57, 58] is a domain-specific language for image processing that allows the separation of the algorithm from its schedule, where the algorithm specifies the operations to be executed and the schedule specifies how the generated code must be organized. Schedules in Halide do not influence in the final output of the program, as they are only used for performance purposes.

Halide makes it much easier to write optimized image processing programs because only the schedule has to be adjusted, thought writing good schedules for Halide is not an easy task. To achieve good scheduling of Halide pipelines, it is still necessary to have a deep understanding of the target architectures. Besides that, when the size and complexity of the pipeline grows, finding good schedules becomes much more challenging for the schedule developers.

A profiler is a good tool to help the developer on program optimizations because it allows to identify bottlenecks and hot-spots in a code by providing information about resource usage during the application runtime, *i.e.* by measuring CPU events such as cache misses, float operations and data transfers between cache levels.

In this work we present an extension to the Halide DSL to enhance its profiling capabilities by allowing the instrumentation of the generated code for a given pipeline schedule. We focus on counting CPU performance events during the application runtime using the PAPI Library [61]. This information can be used by experienced programmers to decide on the best schedule for a specific hardware, but can also be used to provide feedback to machine learning driven automatic schedule generation systems [46, 45, 69, 42].

It is worth to mention here that Halide already contains a simple profiler used for helping on auto-scheduling Halide applications [45]. However, at the moment it only uses memory

footprint, time and estimates parallelism based on the average number of threads used. This can lead schedule developers to red hirrings by using the profiler since these measurements can be influenced by other factors such as bandwitdh limitation and poor locality, for example.

The goal of our work is to write a richer profiler for the Halide language. We modify the Halide framework to generate the annotations that must be used by the profiler and then display specific profiler measurements for the different stages in the pipeline. Our specific goals can be expressed as follows:

- Provide a simple extension to the Halide language to allow the instrumentation of specific regions in the generated code.
- Show how our profiler can be useful for scheduling Halide applications by finding application bottlenecks.
- Display the obtained measurements for some Halide applications with different schedules, comparing the expected behavior with the obtained results.
- Emphasize the usage of profilers for tuning applications in both manually tuning and automatic generation of optimized code.

We describe the proposed extension to the Halide language, how the profiling code snippets are inserted into the Halide generated code, its coupling to the PAPI library during runtime and demonstrate its capabilities by counting L1 cache misses, the number of floating-point operations (FLOP) and the L3 cache data volume on four different schedules for the image *Blur* pipeline.

The first part of this dissertation explains about image processing and its importance today, and also shows profiler technologies and available domain-specific languages for code optimization. In the second part we focus on the Halide framework, showing how it works and how Halide can be useful for writing high performance image processing code for different architectures. The third part we focus on showing state-of-the-art technologies for code profiling, and also present some work on automatic tuning strategies using profiled events to motivate the reader with our contribution. Next, we explain how our extension can be used, and how it was implemented. Finally, we expose some experimental results obtained with our profiler and discuss how these results can help on finding schedules with better performances.

2 THEORETICAL FRAMEWORK

2.1 IMAGE PROCESSING

Image processing is a sub-field of digital signal processing that deals with the processing of analog and digital images. The image processing area has potentially evolved in the last years due to the improvement of artificial intelligence algorithms performance and the presence of smart-phone devices. Pattern recognition, MRI processing [52] and image reconstruction are showing impressive results using machine learning techniques together with image processing and computer vision algorithms. On the other hand, studies for executing image processing with machine learning on smart-phones are also showing good progress [27].

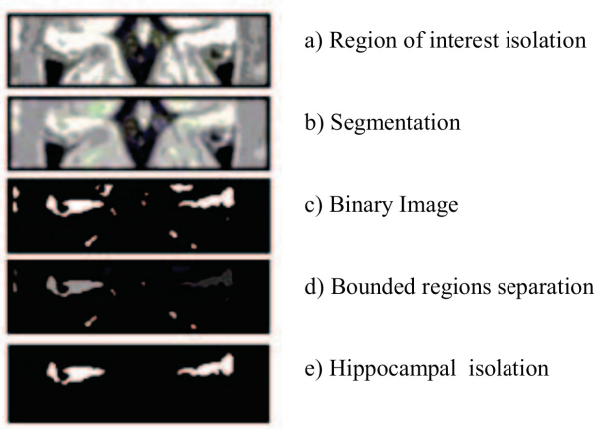


Figure 2.1: Hippocampal isolation using image processing techniques. (Source: [52])

In the era of digital computers, several devices contain cameras with optical sensors for capturing image signals and decode them from raw data into digital images (also removing several noise contained in these raw data). These devices must also be able to perform from basic to more complex operations on these images for image enhancement and sharpening, as well as doing feature extraction on input images for obtaining precise information about them. Computer vision algorithms commonly used for bio-medical image processing [5, 18], pedestrian tracking [74, 17, 68], face and vehicle plate recognition [75, 6, 25], image segmentation [63] and many other purposes also rely on machine learning and image processing algorithms for improving their efficiency by reducing noise, restoration or even improving the acquisition of the images.

However, with the quality of digital images growing up due to the improvement of optical sensors on capturing much more raw data and the presence of tridimensional images, reaching good performance for image processing algorithms is becoming a much more difficult and time-consuming task. Also, the presence of more sophisticated and heterogeneous hardware launch a new challenge on keeping up the algorithms performance with state-of-the-art processors and accelerators.

Image processing algorithms usually present memory intensive operations which requires locality improvement for not being penalized by accesses on slower memories. The reorganization of the image processing codes nowadays must be performed for each individual present architecture, and as it was not enough, target devices with the same architectures can present very different configurations that may lead to the device not taking advantage of the performance optimizations for a specific code variant.

The GPU devices are well-suited hardware for image processing applications because they are designed towards SIMD (Single Instruction, Multiple Data) parallelism, which executes the same instruction at multiple data in parallel and can efficiently execute the majority of existing image processing algorithms. Applications that must execute on GPU devices can be written using available programming models such as CUDA [49] and OpenCL [65]. Nevertheless, writing and tuning code for GPU devices is a very error-prone task and the amount of work and knowledge necessary is much higher compared to writing a single code for running on general purpose processors. Also, even on general purpose processors, tuning image processing algorithms may require a good understanding of the target architecture, like the use of vectorization using AVX/SSE instructions, which needs to be implemented in low-level assembly language. Demands for better ways of programming image processing applications that emits optimized code for several architectures are often studied to find solutions on these issues.

The Halide [58] framework attempts to solve this problem by using a domain-specific language that describes image processing pipelines, and then separately writing a schedule for them. This has the advantage that image processing algorithms are often implemented and optimized individually by steps (on which each step would be a filter for applying in the image), this limitation is caused due to the execution strategy be coupled to the algorithm logic on general purpose programming languages. Since Halide separates the algorithm from the strategy, it allows the optimization across different stages since the schedule can specify that a filter B could be applied on the same iteration of a filter A application, maximizing locality between them. However, writing good schedules for Halide algorithms still requires a good understanding of the target architectures and code optimization.

2.2 DOMAIN SPECIFIC LANGUAGES FOR CODE OPTIMIZATION

Many efforts have been made for generating optimized code for several architectures by using a high-level representation of the program. Since the space of possible programs that can be written in a general purpose programming language is infinitely high, works are usually restricted to a specific domain of problems like stencil computation, image processing or partial differential equation solvers by the multigrid method, for instance.

Domain-specific languages can be implemented as *external* separate languages or be *embedded* into another existing language [64]. External DSLs give more flexibility on defining the language syntax, and also it is easier to perform error-reporting on external separate languages. However, engineering costs for external languages are higher, because it is necessary to implement all the compiler steps (lexer, parser, semantic rules, etc...) for the language.

Embedded languages cost less to be developed because they are implemented on top of a *host language*, thus its primitives are written in the host language. Domain-specific languages can be shallowly- or deeply-embedded into a host language. Shallow embedding implements the DSL primitives to compute values in the host language, whereas deep embedding implements the DSL primitives to produce an in-memory structure (intermediate representation) of the program. Shallow embedding uses the host language compiler to produce native code, whereas deep embedding compiles the intermediate representation structure to produce native code.

Countless domain-specific languages are designed using one of the three approaches (external, shallowly embedded or deeply embedded). External languages have been losing space to embedded languages as they are harder to implement and maintain. Shallow embedding DSLs tend to be easier to implement than deep embedding, but since it uses the host language compiler, optimizations are in general more restricted on shallowly embedded DSLs. Additionaly, runtime

optimizations are not allowed in shallow embedded DSLs, seeing that it uses the compiler of the host language to generate programs.

Using a domain-specific language allows front-end domain-specific developers to write in a more friendly way the operations that must be performed on their application, while the execution strategy is defined by the compiler. The compiler in this case may be developed by a machine expert which has much more experience on tuning code for a specific architecture. This way, the domain-specific language implementation can be compiled into a low-level, optimized representation for a specific architecture such as x86 or CUDA, for example.

Another approach is using a domain-specific language that separates the algorithm from the strategy, this can usually be achieved by using higher-order functions on shallowly-embedded languages. In this case, the application developer writes the front-end implementation (which also happens in the previous approach), and another code representing the strategy to be performed is written by the machine expert. Using the algorithm and the schedule defined as inputs, the compiler should be able to generate the low-level representation for a specific architecture. This approach usually leaves the optimization step in the definition of the strategy, and the compiler just generates the application code with the defined schedules. This approach also allows experimentation of different strategies for different architectures, as well as the use of meta-heuristics for finding the best schedule.

PATUS [11] is a code generation and auto-tuning tool for the class of stencil computation problems. PATUS uses a domain specific language for perfomance and experimentation purposes using different parallelization and optimization strategies on different target architectures. PATUS separates its applications between the stencil and the strategy, which is very similar to Halide concepts of algorithm and schedule.

The stencil specification contains the stencil computation characteristics such as the boundary treatment and the grid traversal (e.g. Jacobi, Gauss-Seidel, colored iterations). Strategies describe the parallelization methods or bandwidth saving algorithms by using a second domain specific language. The strategy is independent of the stencil and the hardware architecture used. PATUS assumes that the order in which individual grid points execute does not change the result of the entire computation, this way, the order is not preserved across different strategies.

SPIRAL [55] is a platform for the generation of high performance code for the domain of linear digital signal processing (DSP). SPIRAL explores the mathematical structure of the algorithm domain, generating many alternative algorithms for a given transform and then uses search and learning techniques for exploring the space of these alternatives to find the one that is the most suitable for the chosen platform.

Following this chapter, we describe some state-of-the-art domain-specific languages that generate optimized code for different architectures, as well as AnyDSL [31, 34], a framework for writing domain-specific libraries through the use of higher-order functions.

2.2.1 AnyDSL

AnyDSL [31, 34] is a framework for writing domain-specific libraries by means of higher-order functions, its main contribution is Thorin [35], a Continuation-Passing Style (CPS) [2, 1] IR that performs optimizations and reduces the overhead caused by this approach. AnyDSL attempts to ease the implementation of domain-specific languages by splitting work between three different areas: (a) the application developer, which just uses the DSL, (b) the DSL designer, encharged by

the definition of the language elements related to the domain, and (c) the machine expert, which is responsible for optimizing the code for one or more specific target architectures.

Since it is a framework for writing domain-specific libraries, AnyDSL is not restricted to a domain. Until the present moment, AnyDSL has already been used for optimizing ray-tracing applications using Bounding Volume Hierarchy (BVH) ray traversal algorithms [54], refining multigrid application for different targets [40] and stencil computation refinement [32].

AnyDSL uses Impala, a dialect of Rust, as its front-end language. This means that domain-specific libraries in AnyDSL are shallowly-embedded in the Impala language, using its compiler and partial evaluator to emit native code. Listing 2.1 shows the part of the application that must be written by the application developer. The code is a very simple code that just calls the image processing domain-specific functions **load** and **gaussian_blur**. Impala codes can be linked with C/C++ functions since they can be compiled into LLVM IR, this way, the **load** function could ended up calling a function in C/C++ that loads an image from a file (such as the OpenCV implementation for loading an image from a file, for instance). This allows the reuse of existing implementations not necessarily optimized through the AnyDSL model.

Listing 2.1: Impala code written by the application developer.

```
1 fn main() {
2     let img = load("dragon.png");
3     let blurred = gaussian_blur(img);
4 }
```

Listing 2.2 shows the implementation of the **gaussian_blur** function in Impala, which must be written by the DSL designer. Notice the use of the higher-order function **iterate**, which specifies how to iterate over the **out** field. The **iterate** function is considered a higher-order function because it receives a function as a parameter, which in this case is the body of the **for** statement. The **for...in** parameter in Impala is a syntactic sugar for calling a higher-order function with the body code as its parameter function. The free variables are defined between the **for...in** operator (in this case, **x** and **y**).

Listing 2.2: Impala code written by the DSL designer.

```
1  fn gaussian_blur(field: Field) -> Field {
2      let stencil: Stencil = { /* ... */ };
3      let mut out: Field = { /* ... */ };
4
5      for x, y in @iterate(out) {
6          out.data(x, y) = apply_stencil(x, y, field, stencil);
7      }
8
9      out
10 }
```

The **iterate** function can be implemented for distinct architectures by defining different values for the free variables. Listing 2.3 shows an implementation for generating NVVM IR code (NVIDIA IR based on LLVM IR that represents GPU compute kernels), this code usually must be written by the machine expert. In the code, it is also noticeable the presence of target-specific functions, such as **nvvm** that uses the grid and block configurations for executing a kernel in a GPU, as well as the functions used for the **x** and **y** calculations. The body function abstracts the logic written by the DSL designer, so the machine expert only has to worry about how to iterate over the field, which is basically the strategy to execute the algorithm.

Listing 2.3: Impala code written by the machine expert.

```
fn iterate(field: Field, body: fn(int, int) -> ()) -> () {
    let grid = (field.cols, field.rows, 1);
    let block = (128, 1, 1);

    with nvvm(grid, block) {
        let x = nvvm_tid_x() + nvvm_ntid_x() * nvvm_ctaid_x();
        let y = nvvm_tid_y() + nvvm_ntid_y() * nvvm_ctaid_y();
        body(x, y);
    }
}
```

A possible version of the **iterate** function that could be used for a x86 CPU target could be written by iterating **x** and **y** through simple loops over the field dimensions and then calling the body function. A probably better version could be written by using the *loop blocking* technique for tiling the field domain over blocks for maximizing locality.

Another important thing to look at Listing 2.2 is the @ symbol. This symbol triggers the Impala partial evaluator in the specified function, which will aggressively perform specialization of the program by precomputing static values at compile time. This should produce a residual program that is generally faster than the original one. The partial evaluator is recursively triggered when the @ is used in a function (this means that all code, including calls within the `iterate` function will also be specialized). To disable the partial evaluator for a specific region of code, the $ symbol can be used.

This approach of using higher-order functions has significant overhead on common imperative languages because several transformations such as *closure-conversion* must be used for representing them in lower representations. Since Impala is compiled to Thorin IR, this overhead is practically ceased. This happens because Thorin is a functional representation, and functional representations intrinsically express higher-order functions. That is why Thorin is a fundamental part of the AnyDSL model.

2.2.2 ExaSlang

ExaSlang [62] is a domain-specific language by ExaStencils [19] used for solving partial differential equations (PDEs) by the Multigrid method. ExaSlang compile specifications at a high abstraction level to produce low-level optimized and highly-scalable code in C++ using OpenMP [51] and MPI [44].

ExaSlang compiler is written in Scala and uses parser combinators to facilitate the development of parsers [29, 28]. Also, ExaSlang follows a multi-layer approach on which each layer represents a different abstraction level for code writing. On the first layer, for example, the user just specifies the partial differential equation he wants to solve and ExaSlang generates the optimized code for the chosen target. On the last layer (called Complete Program Specification), functional programming elements are used for writing the code for solving the PDE.

2.2.3 PolyMage

PolyMage [46] is a domain-specific language and a compiler for image processing pipelines. PolyMage uses Halide approach for performing optimizations across stages in the pipeline, however it does not rely on a schedule specification for generating the high performance implementations, instead it uses an optimizer and an autotuner for it.

The PolyMage domain-specific language is embedded in Python and it is inspired by Halide. Figure 2.3 shows the specification of Harris corner detection [24] in PolyMage and also illustrates the pipeline for the specification.

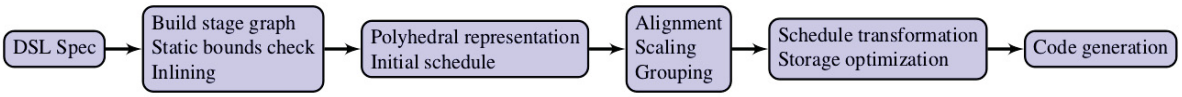


Figure 2.2: Steps during PolyMage compilation. (Source: [46])

Figure 2.2 shows the sequences of PolyMage compiler phases. In the first phase, the DAG is built, check for static bounds and inlining is performed. Each function in a PolyMage pipeline is either a point-wise, stencil or sampling operation. Inlining of functions improves locality in exchange for redundant computations and, for simplicity purposes, it is only performed for point-wise operations in PolyMage. Other operations rely on schedule transformations for achieving better locality.

```
1   R, C = Parameter(Int, "R"), Parameter(Int, "C")
2   I = Image(Float, "I", [R+2, C+2])
3
4   x, y = Variable("x"), Variable("y")
5   row, col = Interval(0, R+1, 1), Interval(0, C+1, 1)
6
7   c = Condition(x, '>=', 1) & Condition(x, '<=', R) &
8      Condition(y, '>=', 1) & Condition(y, '<=', C)
9
10  cb = Condition(x, '>=', 2) & Condition(x, '<=', R-1) &
11      Condition(y, '>=', 2) & Condition(y, '<=', C-1)
12
13  Iy = Function(varDom = ([x, y], [row, col]), Float, "Iy")
14  Iy.defn = [ Case(c, Stencil(I(x, y), 1.0/12,
15                              [[-1,-2,-1],
16                               [ 0, 0, 0],
17                               [ 1, 2, 1]])) ]
18
19  Ix = Function(varDom = ([x, y], [row, col]), Float, "Ix")
20  Ix.defn = [ Case(c, Stencil(I(x, y), 1.0/12,
21                              [[-1, 0, 1],
22                               [-2, 0, 2],
23                               [-1, 0, 1]])) ]
24
25  Ixx = Function(varDom = ([x, y], [row, col]), Float, "Ixx")
26  Ixx.defn = [ Case(c, Ix(x, y) * Ix(x, y)) ]
27
28  Iyy = Function(varDom = ([x, y], [row, col]), Float, "Iyy")
29  Iyy.defn = [ Case(c, Iy(x, y) * Iy(x, y)) ]
30
31  Ixy = Function(varDom = ([x, y], [row, col]), Float, "Ixy")
32  Ixy.defn = [ Case(c, Ix(x, y) * Iy(x, y)) ]
33
34  Sxx = Function(varDom = ([x, y], [row, col]), Float, "Sxx")
35  Syy = Function(varDom = ([x, y], [row, col]), Float, "Syy")
36  Sxy = Function(varDom = ([x, y], [row, col]), Float, "Sxy")
37  for pair in [(Sxx, Ixx), (Syy, Iyy), (Sxy, Ixy)]:
38     pair[0].defn = [ Case(cb, Stencil(pair[1], 1,
39                                 [[1, 1, 1],
40                                  [1, 1, 1],
41                                  [1, 1, 1]])) ]
42
43  Det = Function(varDom = ([x, y], [row, col]), Float, "det")
44  d = Sxx(x, y) * Syy(x, y) - Sxy(x, y) * Sxy(x, y)
45  Det.defn = [ Case(cb, d) ]
46
47  Trace = Function(varDom = ([x, y], [row, col]), Float, "trace")
48  Trace.defn = [ Case(cb, Sxx(x, y) + Syy(x, y)) ]
49
50  Harris = Function(varDom = ([x, y], [row, col]), Float, "harris")
51  coarsity = Det(x, y) - 0.04 * Trace(x, y) * Trace(x, y)
52  Harris.defn = [ Case(cb, coarsity) ]
```

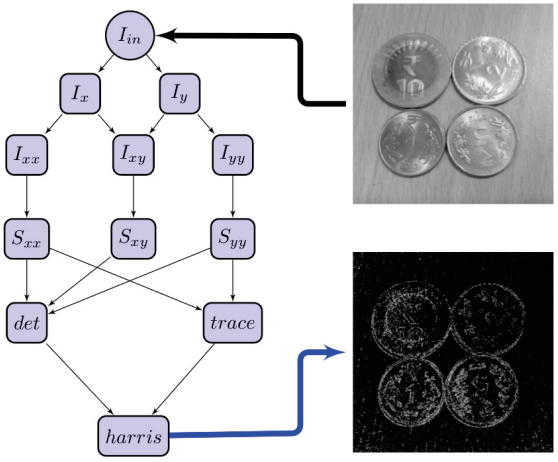


Figure 2.3: Harris corner detection specification in PolyMage is shown at left and its pipeline DAG is shown at right. (Source: [46])

Next, the compiler uses the polyhedral model [22, 7], a mathematical framework for representing schedules and generating their respective loop nests. The polyhedral framework helps on dealing with complex transformations and enables precise dependence analysis. Using the polyhedral model requires a polyhedral representation of the schedule. Considering the **harris** function domain definition as exposed by Mullapudi et al [46]:

$$harris_{dom} = \{\, (x, y) \mid x \geq 2 \wedge x \leq R - 1 \wedge y \geq 2 \wedge y \leq C - 1 \,\}.$$

The definition of a schedule in the polyhedral domain can be defined by a relation between two integer sets, on which the domain corresponds to the function domain and the range gives the ordering for evaluating the function. The example below [46] shows a schedule for the **harris** function:

$$harris_{dom} = \{ (x, y) \rightarrow (y, x) \mid x \geq 2 \wedge x \leq R - 1 \wedge y \geq 2 \wedge y \leq C - 1 \}.$$

```
for y in [2 ... C-1]:
    for x in [2 .... R-1]:
        harris(x, y)
```

The order in the above case is given by the multi-dimensional time stamp (y, x), the schedule is then given by its lexicographic ordering as shown in the pseudo-code. Alternatively, schedules in the polyhedral model can be described using hyperplanes which provide better geometric intuition on dealing with tiling transformations [46].

The compiler then builds an initial schedule from the pipeline specification using the topological order and the domain variables of each function, and then extract the dependence information from this schedule. The dependency information is obtained through *dependence vectors* which are calculated by subtracting the time stamp on which a value is produced from the time stamp which it is consumed.

PolyMage uses overlapped tiling [26, 30] technique since it is shown to be the most suitable choice for image processing pipelines. However, since the mentioned techniques [26, 30] for overlapped tiling are developed towards time-iterated stencil dependence patterns, schedule transformations must be applied for PolyMage's complex and heterogeneous pipelines. Alignment and scaling of functions is applied for turning dependence vectors constant for functions, which is necessary for building the overlapped tiles.

After the transformations, functions are grouped through an heuristic that uses the scaling and alignment factors in consideration, the pipeline is then partitioned into groups. Next, the compiler determines the tile shape for each dimension by looking into the dependence vectors. The tile shape is determined by comparing dependence vectors between two levels of the pipeline DAG. Since the overlapped tile is applied for entire group of stages, the compiler then modifies the schedules for all the functions in the group.

Finally, the compiler performs storage optimization by using scratchpad allocation for intermediate functions, also remapping the accesses to the scratchpad region. The compiler then generates a C++ function implementing the pipeline using OpenMP [51] for turning parallel the outermost parallel dimension in each group, as well as the integer set library (isl) [67] for generating loops that scan each group of functions in the schedule.

2.2.4 HIPAcc

HIPA^{cc} [39] is a domain-specific language and a source-to-source compiler for the image processing domain. HIPA^{cc} uses C++ templates for defining a DSL embedded in C++ to describe image processing algorithms. The compiler then uses the DSL specification for generating C++ code with target-specific optimizations. HIPA^{cc} uses CUDA and OpenCL for emitting efficient code for GPU, Renderscript [59] for generating code for Android mobiles and also can generate FPGA-based image processing accelerators through HLS [47] with *Xilinx Vivado HLS* and *Altera*

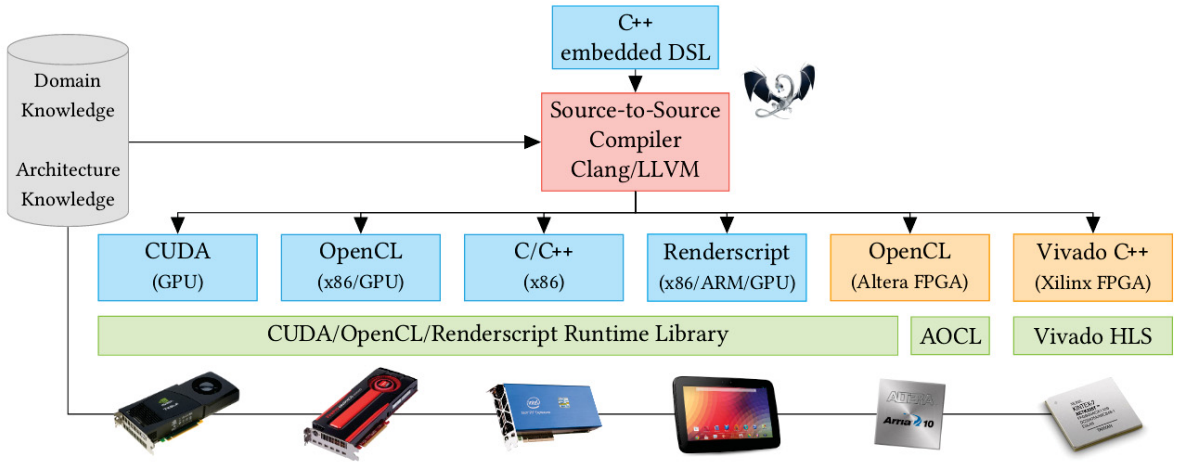


Figure 2.4: Overview of the HIPA^{cc} framework. C++ embedded DSL code is compiled through a source-to-source compiler that uses Clang/LLVM. The code is compiled to target-specific code using one of the available programming models (CUDA, OpenCL, intrinsic C/C++ code for x86, Renderscript, OpenCL with Altera SDK or Vivado C++). (Source: [60])

SDK for OpenCL [60]. Figure 2.4 [60] shows an overview of the HIPA^{cc} framework as well as its target architectures.

Listing 2.4 shows the implementation of a linear filter kernel using HIPA^{cc}. All kernels are defined as a subclass inherited from the **Kernel** class. Kernels are the main part of HIPA^{cc} programs and practically all optimizations are performed in the kernel region.

Listing 2.4: Linear filter kernel written using the HIPA^{cc} DSL. (Source: [38])

```
1  class LinearFilter : public Kernel<uchar4> {
2   private:
3     Accessor<uchar4> &input;
4     Mask<float> &mask;
5
6   public:
7     LinearFilter(IterationSpace<uchar4> &iter,
8                  Accessor<uchar4> &input,
9                  Mask<float> &mask)
10       : Kernel(iter), input(input), mask(mask) {
11      addAccessor(&input);
12    }
13
14    void kernel() {
15      float4 sum =
16          convolve(mask, Reduce::SUM, [&] () -> float4 {
17            return mask() * convert_float4(input(mask));
18          });
19      output() = convert_uchar4(sum + 0.5f);
20    }
21 };
```

Besides the kernel, HIPA^{cc} uses specific classes for instantiation of masks (filters), images, boundary conditions, accessors and iteration spaces. These instantiations are then mapped to target-specific runtime code in the generated code. HIPA^{cc} uses LLVM with Clang compiler for compiling the C++ input code with the DSL references into an AST representation of the program. HIPA^{cc} then walks into the AST for replacing runtime functions and generating the optimized kernel code.

2.3 PROFILERS

A profiler is a tool that extracts features and measures resource usages from a specific piece of code inside a program. These measurements are usually performed by means of register counters, kernel traps or estimates using already known metrics. Profilers often uses platform-specific code and resources to obtain its measures which not only hinders its portability but also restricts certain measurements to only a subset of its target hardware.

Register counters measurement is performed through specific registers available in the target hardware, these registers work by counting events every time they occur. An example of register counter measurement is the counting of branch miss-predictions in the processor. If a register is set to count branch miss-predictions, every time the processor miss-predicts a branch (assume it as taken and it is not taken or assume it as not taken and it is taken), the value of this register is incremented. Register counters are specific to target hardware and usually allows to extract more specific and low-level features from it.

Kernel traps are used for obtaining low-level software measurements, it involves the injection of code on specific parts of the operating system that must be necessarily called during the execution of the events that must be measured. The most common examples are the system calls (*syscalls*) available in the operating system. The injected code works as a reporter of the feature that must be measured and another programmatically counter can be used for accumulating the reported values in a specific fraction of the time. An example in this case could be the trapping of the *syscall* **write**, the injected code can always jump to another region of code that keeps accumulating the parameter that specifies the number of bytes written and so it is possible for obtaining the number of bytes written in the disk, for example.

Estimates are another way of measuring resources, it involves using a specific event or piece of code that is directly related to the resource we want to measure. Estimates can be used for features that are not too simple to determine accurately or that can be efficiently determined through estimate and it is worth to sacrifice precision in favor of complexity or speed (if the estimate takes less time than the other way around). An example of estimate is footprint measurement of pieces of code, since it is not easy to determine exactly how much memory a specific part of code uses, it is possible for estimating it by using the data structures declared on its scope.

2.3.1 Perf

The perf tools (also known as perf_events) is a Linux kernel subsystem that provides an interface for performance analysis and troubleshooting functions. Perf allows the access to hardware level and software level features by means of register counters, trace points and software counters.

Perf can be used for profiling, counting events and tracing at level of processor cores, tasks and workloads. Profiling in perf can be executed by means of annotations in the code and sampling. Besides this, perf_events also contains a tool for reporting the data that helps with the analysis of its generated samples.

Perf also provides interface to kprobes and uprobes for kernel-level and user-level dynamic tracing, respectively. The use of kprobes and uprobes can be used for debugging and for counting events with kernel traps since the probes are performed through additional instructions in kernel and user specific functions.

Since its core is implemented in the Linux kernel, perf is a lightweight profiler that can measures events with much more accuracy than many profiling tools available today. This advantage is much more significant when referring to low-level software measurements.

Instead of just focusing on the tools, we can also take advantage of the perf_events modifications in the kernel that permit us to profile parts of the code in a programmatically way. This is possible due to the presence of the **__NR_perf_event_open** system call in the kernel. Listing 2.5 shows an example of using perf_events through kernel system calls.

Listing 2.5: Example usage of perf_events in C code through system calls.

```
1  static long perf_event_open(
2    struct perf_event_attr *hw_event, pid_t pid,
3    int cpu, int group_fd, unsigned long flags
4  ) {
5    return syscall(__NR_perf_event_open, hw_event, pid, cpu,
6                   group_fd, flags);
7  }
8
9  int main(int argc, char **argv) {
10   struct perf_event_attr pe;
11   long long count;
12   int fd;
13
14   memset(&pe, 0, sizeof(struct perf_event_attr));
15   pe.type = PERF_TYPE_HARDWARE;
16   pe.size = sizeof(struct perf_event_attr);
17   pe.config = PERF_COUNT_HW_INSTRUCTIONS;
18   pe.disabled = 1;
19   pe.exclude_kernel = 1;
20   pe.exclude_hv = 1;
21
22   fd = perf_event_open(&pe, 0, -1, -1, 0);
23
24   ioctl(fd, PERF_EVENT_IOC_RESET, 0);
25   ioctl(fd, PERF_EVENT_IOC_ENABLE, 0);
26
27   printf("Measuring instruction count for this printf\n");
28
29   ioctl(fd, PERF_EVENT_IOC_DISABLE, 0);
30   read(fd, &count, sizeof(long long));
31
32   printf("Used %lld instructions\n", count);
33
34   close(fd);
35 }
```

The code uses the perf_event_open syscall to count the number of instructions executed during the execution of the **printf** function. The result is then displayed after it is read from the file descriptor.

2.3.2 PAPI

PAPI (Performance API) [61] is a framework that provides a common interface for using performance counters for a variety of microprocessors. It has C and Fortran APIs that allow preset and native events to be measured from various sources. PAPI also supports the use of components (referred as PAPI-C) [13], allowing to measure more performance events than those available in the CPU.

An usage example of PAPI-C [3] is when one wants to measure events from a CUDA GPU. To achieve this, it is necessary to compile PAPI with the CUDA component. PAPI can

also be used for measuring energy and power consumed during the execution of a region of code through the use of specific components [70]. An example of this kind of component is the Running Average Power Limit (RAPL) component, used for measuring power and energy on Intel processors that support that feature [12].

PAPI is designed in two separate layers (as shown in Figure 2.5): the portable layer specifies the parts that are independent of the target machine, more specifically the low level and high level APIs; and the hardware specific layer specifies the interface between the hardware independent functions and the hardware dependent functions that use assembly code, operating system calls or kernel extensions for accessing the hardware counters.

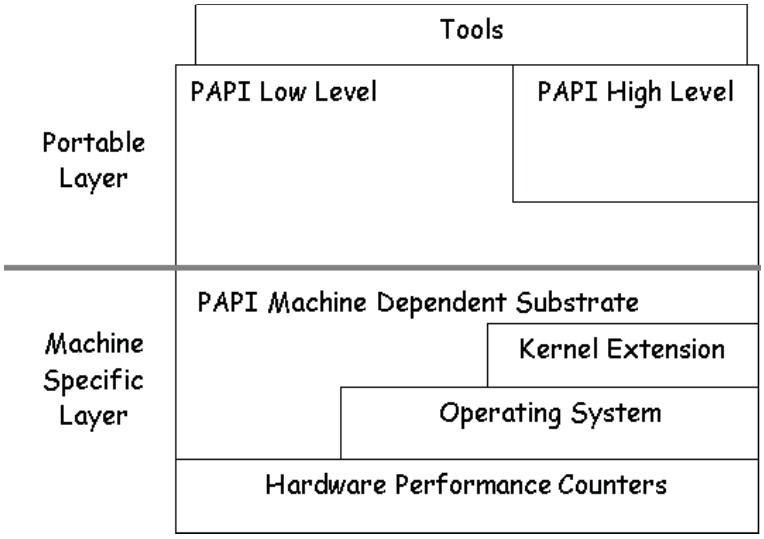


Figure 2.5: PAPI architecture. (Source: [3])

Register counters can be accessed through PAPI's high level and a low level API. The high level API is provided for simplicity purposes and provides limited PAPI measures (such as number of FLOP, cycles and instructions). The low level API can be used for developers who want more fine-grained measurement and control of the PAPI interface, as it allows the specification of event sets that define the events to be measured on the target hardware. The low level API also allows the access through all the PAPI components and native events available to specific hardware.

Listing 2.6 shows an example of code that uses the PAPI high level API for measuring the number of floating point instructions from a specific code abstracted by the **your_slow_code** function call. The example uses the **PAPI_flips** function for both starting and stopping of the register counters.

Listing 2.6: PAPI high level API example.

```
int main() {
  float real_time, proc_time, mflips;
  long long flpins;
  float ireal_time, iproc_time, imflips;
  long long iflpins;

  PAPI_flips(&ireal_time, &iproc_time, &iflpins, &imflips));

  your_slow_code();

  PAPI_flips(&real_time, &proc_time, &flpins, &mflips));

  printf(
    "Real_time: %f Proc_time: %f Total flpins: %lld MFLIPS: %f\n",
    real_time, proc_time, flpins, mflips
  );
}
```

Listing 2.7 shows an example of code that uses the PAPI low level API. The code not only needs to initialize the PAPI library through the **PAPI_library_init** function, as it also needs to create an event set to specify the events to be measured. The example also shows the usage of the **PAPI_reset** and **PAPI_accum** functions, used for ignoring a portion of the code during the profiling. Another function that is shown in the example is the **PAPI_query_event**, which returns if a specific event is available for the target machine or not.

Listing 2.7: PAPI low level API example.

```
int main(int argc, char **argv) {
  long long values[NUM_EVENTS];
  int Events[2] = {PAPI_FP_INS, PAPI_TOT_CYC};
  int EventSet = PAPI_NULL;

  PAPI_library_init(PAPI_VER_CURRENT);

  PAPI_create_eventset(&EventSet);
  PAPI_add_events(EventSet, (int *) Events, NUM_EVENTS);
  PAPI_start(EventSet);

  do_flops(NUM_FLOPS);

  PAPI_read(EventSet, values);
  do_flops(NUM_FLOPS); // Should not be counted
  PAPI_reset(EventSet);
  do_flops(NUM_FLOPS);
  PAPI_accum(EventSet, values);

  printf("%d %d\n", values[0], values[1]);
  return 0;
}
```

Advanced profiling features such as the multiplexing of events through time-sharing (used due to restrictions on the amount of registers on the target processor), measuring parallel programs and overflow handling are also supported. These features are very helpful to profile complex parallel programs and to deal with hardware specific limitations. In the proposed Halide

extension we use PAPI because it can be used on a large variety of platforms, which can help to evaluate our profiler targeting other operating-systems and hardware in the future.

2.4 CONCLUSION

We show the current scenario for the image processing domain and the importance and challenges on achieving good performance applications on heterogeneous hardware. The presence of a framework that permits the specification of an image processing algorithm separated from the schedule shows a very good progress on the area, but there is still a lot of work for the automatic generation of tuned image processing code.

Several domain-specific language frameworks targeting different domains are also exposed for showing different techniques for optimized code generation with promising results. With this, we also emphasize the importance of a domain-specific language in the current context, and also covers more opportunities on other domains that can take benefit from a profiler.

Finally, we show state-of-the-art profiler technologies that are capable of counting several types of events on today's available hardware. We also explain different techniques for profiling code and extracting useful information for troubleshooting and finding bottlenecks in any type of application.

3 THE HALIDE PROGRAMMING LANGUAGE

Halide [57, 58, 36] is a domain-specific language for describing image processing and array processing pipelines through a high-level specification which decouples the algorithms from their execution strategy. This means that a Halide program is split into two parts: the algorithm, that specifies *what* to execute, and the schedule, defining the strategy or *how* to execute the algorithm. This separation allows for an easier development of high performance image processing applications for different architectures because only the second part has to be changed in order to perform optimizations in the generated code.

Halide is a deeply-embedded language that uses C++ as its host language. This means that Halide applications are program generators that produce an in-memory representation that is further compiled to native code. Such approximation can be successfully achieved by using some C++ particular features like operation overloading and polymorphism. At some moment, Halide in-memory representations are compiled to LLVM IR [33] during the compilation process, and LLVM ends up generating the native code for the target platform.

The available target platforms for Halide at the moment are [14]:

- **CPU architectures:** X86, ARM, MIPS, Hexagon, PowerPC
- **Operating systems:** Linux, Windows, macOS, Android, iOS, Qualcomm QuRT
- **GPU Compute APIs:** CUDA, OpenCL, OpenGL, OpenGL Compute Shaders, Apple Metal, Microsoft Direct X 12

In general purpose programming languages, changing and optimizing code for image processing and array processing is very cumbersome. Some pipelines may contain hundreds of stages such as the Local laplacian filters implementation that contains 99 stages (most of them stencil computation stages). Scheduling these stages to be computed in the same level to enhance locality requires considerable changes in the loop nests of the code, which costs time and is a very error-prone task. Halide allows developers to easily experiment among different schedules that compute stages within others, which provides much more optimization opportunities specially in large and complex pipelines.

Besides image processing and array applications, Halide has also been used to implement efficient neural network layers. Li et al [36] extends the Halide language to compute gradients of arbitrary Halide applications using reverse-mode automatic differentiation. The work also shows how the automatic gradient computation combined with an automatic scheduler of Halide applications can be used to implement neural network layers that is faster and significantly simpler than the same versions written in C++ and CUDA.

3.1 ALGORITHM DEFINITION

Listing 3.1 shows a 3×3 *Blur* box filter implemented in Halide (both algorithm and schedule) consisting of two steps (functions) of 3×1 filters. The implementation specifies a Halide pipeline with two functions: **blur_x** and **blur_y**, where **blur_x** can be referred as the producer function and **blur_y** can be referred as the consumer function or the output function, *i.e.*, **blur_y** consumes values produced by **blur_x** to produce the final output.

Listing 3.1: Halide implementation of a 3x3 *Blur* box filter

```
1  Buffer<float> input = Tools::load_and_convert_image("input.png");
2  Buffer<float> output(
3    input.width() - 2, input.height() - 2, input.channels()
4  );
5  Func blur_x, blur_y;
6  Var x, y, xi, yi;
7
8  // The algorithm - no storage or order
9  blur_x(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;
10 blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + blur_x(x, y+1))/3;
11
12 // The schedule - defines order, locality; implies storage
13 blur_y.tile(x, y, xi, yi, 256, 32).vectorize(xi, 8).parallel(y);
14 blur_x.compute_at(blur_y, x).vectorize(x, 8);
15
16 blur_y.realize(output);
```

Line 1 shows the definition of the input buffer, which uses the Halide tools function `load_and_convert_image()` to read the image from a file and converts it into the Halide buffer object. Line 2 specifies the output buffer using the dimensions from the input image. Lines 5-6 declare the functions and variables used in the pipeline.

Lines 9-10 specify the algorithm definition, which is basically the definition of each stage computation separately. Notice that Halide algorithm implementation basically consists of just functional definitions. All nested loops, buffer allocations and boundaries calculation are automatically generated by the compiler based on the defined schedule and input image. Relations between different functions in Halide are also not explicit, they are automatically inferred by exploring the function definitions. For example, at line 10 the **blur_y** function is defined in terms of **blur_x** calls, which implies that **blur_y** is a consumer function of **blur_x** in the pipeline.

Lines 13-14 specify the schedule for each stage. The `tile()`, `vectorize()` and `parallel()` primitives are used to evaluate the function in tiles, use CPU vectorization instructions at some loop level and parallelize a specific loop level, respectively. The `compute_at()` primitive is used to define the computation level of a function. Loop levels are defined in terms of a function and a variable, since each generated loop in Halide refers to the traversal of a dimension in a function.

Finally, line 16 uses the `realize()` function to perform JIT compilation and executing the generated code. After it, the `output` Halide buffer must contain the blurred version of the input buffer.

We can see at lines 5-6 the presence of some Halide primitives such as `Func` and `Var`. The Halide core primitives and a brief description of them are listed as follows:

- `Func`: Pure Halide function, used for defining image values over a nth-dimensional domain (*e.g.* `blur_x` and `blur_y`).
- `Var`: Free variable in the domain of a function (*e.g.* `x` and `y`).
- `RDom`: A multi-dimension iteration domain, effectively an ordered list of bounded variables.
- `Expr`: Expression that defines the value of a function (*e.g.* `blur_x(x - 1, y) + blur_x(x, y)`).

- `Image`: Reference to a immutable external memory buffer, this can be used like a function in Halide.
- `Param`: Runtime variable parameter, can provide scalar arguments to the algorithm.

Understanding the core primitives are the key to develop Halide algorithms. The fact that Halide algorithms can be described by using just a few primitives reveals how it is a simple and powerful language, specially when compared to general purpose languages on writing image processing applications.

3.2 SCHEDULES

Schedules describe how computations are organized in Halide pipelines independently of the algorithm definition. Every Halide pipeline is modeled as a directed acyclic graph (DAG) [20] representation, where each node is a function and each edge expresses a producer-consumer relation between two functions. The major concern one must deal when scheduling an algorithm is the *order of execution* of the tasks in the DAG [58]. Based on the *order of execution*, schedules also end up defining the amount of redundant computation and the size of the memory buffers allocation used for storing the intermediate results.

A Halide pipeline DAG provides two types of choices that must be taken by the schedule for the organization of computation [58]:

1. Choices of organization *within* each stage (individual node)
2. Choices of organization *across* stages (represented by edge relationships)

For the first set of choices (1), Halide schedules can determine at which granularity to compute each of the stage inputs, at which granularity to store each for reuse, and in what order its domain must be traversed — for simplicity purposes, traversal orders in Halide are constrained to traversal orders that can be trivially expressed in form of loop nests. Parallelism also may be performed by turning one or more loops in the nest parallel.

For the second set of choices (2), Halide schedules can determine the granularity at which values are grouped and interleaved between the producer-consumer relations. Since stage relations are a producer-consumer relationship, every value must be computed and stored by the producer stage before it is used by the consumer, otherwise the schedule is not valid.

We separate both types of choices (1) and (2) to exemplify the schedule directives for individual nodes and producer-consumer relations.

3.2.1 Scheduling Individual Nodes

In order to explain the scheduling for individual nodes, we use the single-stage pipeline example in Listing 3.2. The example is a simple gradient algorithm that just sets the sum of `x` and `y` coordinates to the pixel value. We also include the output provided by the `gradient.print_loop_nest()` function in order to display the loops generated by Halide for the given primitives.

Listing 3.2: Halide gradient algorithm example (Source: [14])

```
1    Func gradient("gradient");
2    gradient(x, y) = x + y;
```

3.2.1.1 *reorder primitive*

From the example, we can perceive that the are two dimensions in the input image that must be traversed — more specifically, x and y axis. There are two different ways of traversing this domain, we could traverse each line in the image in an outer loop, and each column in image in an inner — formally known as *row-major order*. Instead, we can also do the opposite: traverse each column in the image in the outer-loop, and each row in the image in the inner loop — formally known as *column-major order*.

The best order to traverse the image depends on the memory design, dictating which elements of the image are in a contiguous space. By default, Halide will always generate the loop nests starting from the rightest dimensions (outer loops) to the leftest ones (inner loops), corresponding to the *row-major order*. Listing 3.3 shows the generated loops for the default schedule in Halide.

Listing 3.3: Loop nests for the default gradient schedule.

```
1    produce gradient:
2      for y:
3        for x:
4          gradient(...) = ...
```

If we want to walk down the columns instead, we can use the **reorder** primitive to schedule our gradient function. This is possible by inserting the following line of code after our algorithm definition:

```
gradient.reorder(y, x);
```

This specifies that we want to reorder our loop nests, where the first parameter tells that the y dimension loop is the inner, and the x dimension the outtest. Listing 3.4 shows the generated loops after using the **reorder** primitive.

Listing 3.4: Loop nests for the reordered gradient schedule.

```
1    produce gradient_col_major:
2      for x:
3        for y:
4          gradient_col_major(...) = ...
```

3.2.1.2 *split primitive*

Besides reordering, it is also possible to break the loop over a dimension into two nested loops. This is done through the use of the **split** primitive, and it is necessary to specify the *split factor*. The inner loop iterates from zero to the given split factor, and the outer loop iterates from zero to the extent of the specified dimension (*e.g.* image width for the x dimension, image height for the y dimension). The following example can be inserted after the algorithm to split the x axis of the gradient function using a split factor of 2.

```
Var x_outer, x_inner;
gradient.split(x, x_outer, x_inner, 2);
```

Note the presence of two additional variables `x_outer` and `x_inner`. These two additional variables can be used further to perform more powerful schedules, and that is where the advantage of split relies on. For example, we could use the split function to break the domain

into strips in the *y* dimension, and then parallelize at the level of each strip. Listing 3.5 shows the generated loops for splitting the *x* dimension in factor of 2 in our gradient example.

Listing 3.5: Loop nests for the split gradient schedule.

```
1    produce gradient_split:
2      for y:
3        for x.x_outer:
4          for x.x_inner in [0, 1]:
5            gradient_split(...) = ...
```

3.2.1.3 fuse primitive

As oppose to split, we can also fuse two variables to merge two loops into a single one. The generate loop will then iterate from zero to the product of the extents of the fused dimensions. The following example can be inserted after the gradient algorithm to fuse both *x* and *y* variables.

```
Var fused;
gradient.fuse(x, y, fused);
```

Fuse is less important than splitting, but it is also useful. Fusing helps scheduling by allowing the parallelization across multiple dimensions without introducing nested parallelism (parallel loops within parallel loops), this is done by fusing nested parallel loops into a single one and then just turning the fused loop parallel. Nested parallelism is allowed in Halide, but often gives worse performance compared to using a single fused parallel loop. Listing 3.6 shows the generated loops for fusing both the *x* and *y* dimensions in our gradient example.

Listing 3.6: Loop nests for the fused gradient schedule.

```
1    produce gradient_fused:
2      for x.fused:
3        gradient_fused(...) = ...
```

3.2.1.4 tile primitive

A very popular and efficient optimization used in image processing and stencil applications is to evaluate the input at the level of small rectangular tiles. To perform tiled evaluation, we need to introduce two additional outer loops iterating at each tile, and the two innermost loops are adjusted to iterate over the elements of each tile. We can already perform tiled evaluation in Halide with the primitives we already learned, by just splitting both dimensions in factor of the tile dimensions and then reordering the accesses so the loops that iterates over different tiles become the outermost loops. The following schedule primitives after the gradient algorithm can be used to perform a evaluation at tiles with dimension of 4×4 pixels:

```
Var x_outer, x_inner, y_outer, y_inner;
gradient.split(x, x_outer, x_inner, 4);
gradient.split(y, y_outer, y_inner, 4);
gradient.reorder(x_inner, y_inner, x_outer, y_outer);
```

Since this pattern is very common, Halide provides the **tile** shorthand for it:

```
gradient.tile( x, y, x_outer, y_outer,
               x_inner, y_inner, 4, 4 );
```

Listing 3.7 shows the generated loops for evaluating the gradient example at level of tiles with dimension of 4×4 pixels.

Listing 3.7: Loop nests for the tiled gradient schedule.

```
produce gradient_tiled:
  for y.y_outer:
    for x.x_outer:
      for y.y_inner in [0, 3]:
        for x.x_inner in [0, 3]:
          gradient_tiled(...) = ...
```

3.2.1.5 unroll primitive

Besides changing the structure of the loop nests, Halide also allows to mark loops to be unrolled, vectorized or parallelized. This can be done by using the **unroll**, **vectorize** and **parallel** primitives.

Unrolling a loop that executes *n* iterations replaces its loop form in the generated code by *n* statements performing each iteration. Unrolling can be used when multiple iterations of the loop share overlapping data, thus reducing the number of times the shared values are computed and loaded.

To unroll a loop by a specific factor in Halide it is necessary to split the loop using the unrolling factor as the split factor and then use the **unroll** primitive in the inner loop. The following schedule primitives after the gradient algorithm can be used to unroll the *x* dimension loop by a factor of 2:

```
Var x_outer, x_inner;
gradient.split(x, x_outer, x_inner, 2);
gradient.unroll(x_inner);
```

This can also be expressed more concisely as:

```
gradient.unroll(x, 2);
```

3.2.1.6 vectorize primitive

Vectorization of a loop in Halide is similarly expressed as unrolling it, but instead it is necessary to use the **vectorize** primitive. The following schedule primitives after the gradient algorithm can be used to vectorize the *x* dimension loop by a factor of 4 (*i.e* using SSE to compute 4-wide vectors):

```
Var x_outer, x_inner;
gradient.split(x, x_outer, x_inner, 4);
gradient.vectorize(x_inner);
```

Syntactic-sugar for this is also similar to the unrolling:

```
gradient.vectorize(x, 4);
```

Notice that vectorization may not be supported by the target processor for the given factor. In such cases, Halide will not generate the vectorization instructions.

3.2.1.7 *parallel primitive*

Finally, we can use the **parallel** primitive in order to use parallelism for a specific dimension. Parallelism in Halide is performed by using a thread pool, so every time a parallel loop iteration must be executed, it must wait until there are is an available thread to acquire, and after its execution the thread is released. The number of threads can be specified by using the Halide run-time function `halide_set_num_threads()`. The following schedule primitive after the gradient algorithm can be used to turn the *y* dimension loop parallel:

```
gradient.parallel(y);
```

All these primitives can be used in order to change the structure and computation for individual nodes (here we used the `gradient` node as example). In the next steps we will be looking at schedule primitives used for reorganizing the computation between relations of nodes.

3.2.2 Scheduling Nodes Relations

In order to explain the scheduling for relation between nodes, we use the two-stage pipeline example in Listing 3.8. The example is a simple pipeline that contains a producer and a consumer functions. We also include the output provided by the `consumer.print_loop_nest()` function in order to display the loops generated by Halide for the given primitives.

Listing 3.8: Halide producer and consumer algorithm example (Source: [14])

```
1  Func producer("producer_default"), consumer("consumer_default");
2
3  // The first stage will be some simple pointwise math similar
4  // to our familiar gradient function. The value at position x,
5  // y is the sin of product of x and y.
6  producer(x, y) = sin(x * y);
7
8  // Now we'll add a second stage which averages together multiple
9  // points in the first stage.
10 consumer(x, y) = (producer(x, y) +
11                   producer(x, y+1) +
12                   producer(x+1, y) +
13                   producer(x+1, y+1))/4;
```

When scheduling multi-stage pipelines, Halide provides primitives to define at which level to store and compute values for producer functions. Using the example in Listing 3.8, we can change the storage and computation levels for the `producer` function, consequently defining how the computation of `producer` and `consumer` values should interleave, as well as the size of the buffer to store intermediate results generated by the `producer` function.

The default behavior in this case is to fully inline the `producer` computation into the `consumer` loop nests, with no storage of intermediate results. Listing 3.9 shows the loop nests generated by the default schedule:

Listing 3.9: Loop nests for the default producer-consumer algorithm schedule.

```
1    produce consumer_default:
2      for y:
3        for x:
4          consumer_default(...) = ...
```

Note that there are no `producer` function reference in the generated loop nests. This happens because the `producer` function was fully inlined in the `consumer` function as

mentioned, so all of its values are computed on the fly as required during the computation of the `consumer` function values.

3.2.2.1 compute_root primitive

Another possibility is to compute all values of the `producer` function, store them into a buffer and then using the produced values to compute the `consumer` function. This can be performed by adding the following scheduling primitive after the algorithm definition:

```
producer.compute_root();
```

The **compute_root()** primitive tells Halide that this `producer` function must be computed at root level (*i.e.* it cannot be computed inside any function). This requires an intermediate buffer with enough size to store all the computed values by the `producer` function, which is a size in the order of the number of pixels in the image. Listing 3.10 shows the loop nests generated when computing the producer function values at the root level:

Listing 3.10: Loop nests for computing the producer values at root level.

```
1   produce producer_root:
2     for y:
3       for x:
4         producer_root(...) = ...
5
6   consume producer_root:
7     produce consumer_root:
8       for y:
9         for x:
10          consumer_root(...) = ...
```

3.2.2.2 compute_at primitive

Besides computing `producer` values at the root level or fully inlining it. It is also possible to perform a midterm choice, an example of this is to perform the computation of `producer` values on each scanline of the `consumer` function. Using interval analysis in the algorithm it is possible to perceive that two scanlines of `producer` values must be computed for each consumer value, therefore this schedule requires a storage buffer with the size of two scanlines for the intermediate values. This can be performed by adding the following scheduling primitive after the algorithm definition:

```
producer.compute_at(consumer, y);
```

Using the `compute_at()` primitive, we determine that the `producer` values must be computed when the `y` variable varies — or, to put differently, it must be computed in the `y` dimension traversal loop. Listing 3.11 shows the loop nests generated when computing the producer function values at the `y` variable level:

Listing 3.11: Loop nests for computing the producer values at the y variable level.

```
produce consumer_y:
  for y:
    produce producer_y:
      for y:
        for x:
          producer_y(...) = ...

    consume producer_y:
      for x:
        consumer_y(...) = ...
```

Defining at which loop level to compute a function allows the schedule developer to perform several optimization across stages. For instance, it is possible interleave the computation of `producer` and `consumer` function at level of tiles. This can be done by first using the **tile** primitive in the consumer function and then defining the computation level of `producer` at the tile outer loop with the **compute_at** primitive. Such strategy is not easy to implement and maintain in general purpose languages when dealing with very large and complex pipelines.

3.2.2.3 store_root primitive

Besides the computation level, we can determine the storage level of a function. Storage level defines at which part of the code the buffer must be allocated to store the produced values, thus it also end up determining its size. In the previous schedule we compute and store `producer` values in the `y` variable level. We can also store the computed values at the root level by adding the following scheduling primitive after the algorithm definition:

```
producer.store_root().compute_at(consumer, x);
```

The most important thing to notice here is that previously we always compute two scanlines per `y` iteration at the `consumer` production. Now, since we store `producer` values at root level, Halide folds the storage down into a circular buffer and we can reuse computed values that overlap between `consumer` iterations at the `y` variable level. Listing 3.12 shows the loop nests generated when computing the producer function values at the `y` variable level and storing them at root level:

Listing 3.12: Loop nests for storing the producer values at root level.

```
store producer_root_y:
  produce consumer_root_y:
    for y:
      produce producer_root_y:
        for y:
          for x:
            producer_root_y(...) = ...

      consume producer_root_y:
        for x:
          consumer_root_y(...) = ...
```

3.2.2.4 store_at primitive

The previous optimization with **store_root()** should deliver very good performance by reusing computed values, so why not always use **store_root()** in our schedules? The problem arises

when parallelism come into play. If we have multiple threads storing values into the circular buffer, then we have a race condition and therefore our code does not produce the correct results. For this reason, Halide prevents this optimization to be done when we turn one of the loops in the nest parallel.

To combine the storage optimization with parallel code, we can use the **store_at()** schedule primitive to set the storage to a specific loop level. Defining a specific loop level for storage allows us to still use the storage optimization at loop levels within parallel regions. For instance, we can do it by adding the following scheduling primitives after the algorithm definition:

```
Var yo, yi;
consumer.split(y, yo, yi, 16);
consumer.parallel(yo);

producer.store_at(consumer, yo);
producer.compute_at(consumer, yi);
```

This schedule will split the domain into strips of 16 scanlines, parallelizing over each strip. Values are stored per strip and computed per scanline inside each strip, allowing values in the same strip to be reused through successive scanlines. Listing 3.13 shows the loop nests generated by the schedule:

Listing 3.13: Loop nests for parallel schedule with storage optimization.

```
1    produce consumer_mixed:
2      parallel y.yo:
3        store producer_mixed:
4          for y.yi in [0, 15]:
5            produce producer_mixed:
6              for y:
7                for x:
8                  producer_mixed(...) = ...
9
10           consume producer_mixed:
11             for x:
12               consumer_mixed(...) = ...
```

To put this in another way, we could not have a storage being performed outside a parallel loop, because this storage would be competed by different threads. Parallelism is then limited by the storage level. By looking at the loop nests it is important to notice that values are defined to be stored only within the parallel loop, which assures the schedule produces the correct output.

3.3 SCHEDULING TRADE-OFFS

Writing a good schedule for a Halide algorithm is not a trivial task, because it requires understanding how the hardware architecture is designed, and how the code reorganization and transformation impacts on its execution. Ragan-Kelley [58] argues that performance on image processing programs is limited by tradeoffs among parallelism, locality and redundant computations. Halide schedules allow to explore different tradeoffs configurations. For a better understanding on how these tradeoffs limit performance, we use the algorithm example from Listing 3.1 and expose four different strategies [58] as depicted in Figure 3.1:

1. Breadth-first: executes the **blur_x** function in the entire image first and then executes the **blur_y** function on the values produced by **blur_x**;

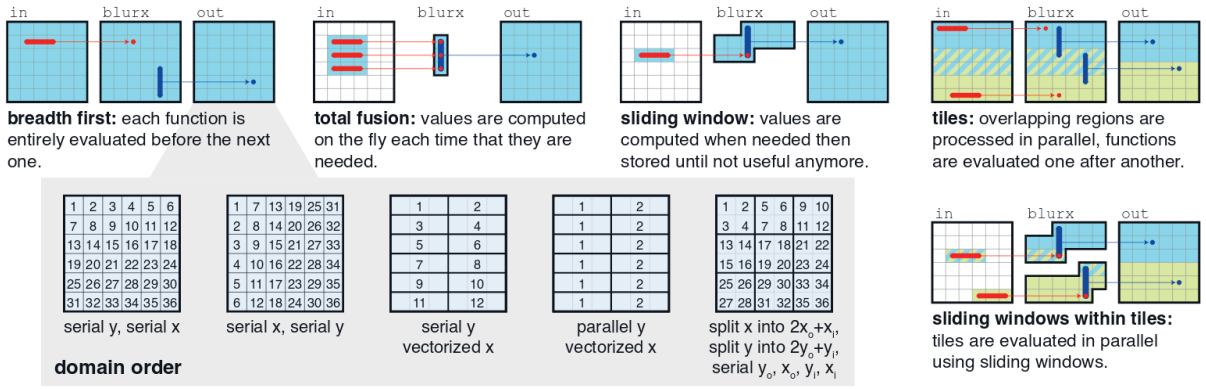


Figure 3.1: Strategies in Halide present different trade-offs between locality, parallelism and redundant computation. The breadth first, total fusion and sliding window are strategies that shows poor locality, high redundant computation and low parallelism, respectively. Tiling and sliding window within tiles are strategies that balances these limitations and so tends to show a much better performance. The strategies also define the domain order traversal for each axis of the input, which can be serial, parallel or vectorized. Domain traversal can also be splitted between blocks (tiled) on which intra-block and inter-block traversal orders can be defined.

2. Full fusion: calculates the **blur_x** for each pixel and in the same iteration performs the **blur_y** computation of that pixel;

3. Sliding window: interleaves the computation of **blur_x** and **blur_y** by storing the values of **blur_x** across uses. This is done using a sliding window which contains the size of three scanlines (three times the width of the image);

4. Tiling: interleaves the computation of **blur_x** and **blur_y** at the level of **tiles**, so that one tile is completely processed before proceding to the next.

The breadth-first strategy performs massive parallel operations as all the pixels are computed independently. However, it has poor locality since it calculates all **blur_x** functions for the entire image and only then starts to execute the **blur_y** function (again, for the entire image). It is most likely that when the breadth-first strategy is computing **blur_y** for a pixel, its region is not in cache anymore, which compromises performance by forcing a load from the hardware slow main memory. The following pseudo-code [58] shows the blur algorithm using the breadth-first strategy:

```
alloc blur_x[2048][3072]
for each y in 0..2048:
  for each x in 0..3072:
    blur_x[y][x] = in[y][x - 1] + in[y][x] + in[y][x + 1]

alloc blur_y[2046][3072]
for each y in 1..2047:
  for each x in 0..3072:
    blur_y[y][x] = blur_x[y - 1][x] + blur_x[y][x] + blur_x[y + 1][x]
```

In contrast to the breadth-first strategy, full fusion has full locality, that is, all **blur_y** functions are performed in the same iteration right after computing **blur_x** for a specific pixel. On the other hand, full fusion requires the computation of **blur_x** three times for the same pixel, while breadth-first requires only one. This redundant computation also affects performance by increasing the number of float operations performed. The following pseudo-code [58] shows the blur algorithm using the full fusion strategy:

```
alloc out[2046][3072]
for each y in 1..2047:
  for each x in 0..3072:
    alloc blur_x[-1..1]
    for each i in -1..1:
      blur_x[i] = in[y - 1 + i][x - 1]
                + in[y - 1 + i][x]
                + in[y - 1 + i][x + 1]
    blur_y[y][x] = blur_x[0] + blur_x[1] + blur_x[2]
```

The sliding window strategy has full locality (as long as the computing window fits into the cache) but does not perform redundant computation, yet it does not perform parallel computations. The sliding window strategy introduces a new dependency during a **blur_y** computation: to execute a **blur_y** on a pixel, all its neighbors must be computed and stored in the sliding window first. This dependency harms parallelism from the code because every **blur_y** computation depends on the sliding window resource. If two **blur_y** operations would execute in parallel, both would get incorrect results, as both would be writing and reading from the same memory region (the sliding window region). The following pseudo-code [58] shows the blur algorithm using the sliding window strategy:

```
alloc blur_y[2046][3072]
alloc blur_x[3][3072]
for each y in -1..2047:
  for each x in 0..3072:
    blur_x[(y + 1) % 3][x] = in[y + 1][x - 1]
                           + in[y + 1][x]
                           + in[y + 1][x + 1]
    if y < 1: continue
    blur_y[y][x] = blur_x[(y - 1) % 3][x]
                 + blur_x[ y % 3 ][x]
                 + blur_x[(y + 1) % 3][x]
```

A better balance for scheduling the mentioned pipeline can be achieved by interleaving the computation of **blur_x** and **blur_y** at the level of **tiles** (also known as overlapped tiling [76, 41]). Redundant computations in this approach occurs in smaller scale as in full fusion, on **blur_x** regions that overlap between tiles, tiles can be processed in parallel because there is no resource dependency between iterations, and locality is achieved as long as the tile region fits in cache, again with memory overhead to load the tile intersection regions. The following pseudo-code [58] shows an example of the blur algorithm using the schedule that performs *tiling*:

```
alloc blur_y[2046][3072]
for each ty in 0..2048/32:
  for each tx in 0..3072/32:
    alloc blur_x[-1..33][32]
    for y in -1..33:
      for x in 0..32:
        blur_x[y][x] = in[ty * 32 + y][tx * 32 + x - 1]
                     + in[ty * 32 + y][tx * 32 + x]
                     + in[ty * 32 + y][tx * 32 + x + 1]
    for y in 0..32:
      for x in 0..32:
        blur_y[ty * 32 + y][tx * 32 + x] = blur_x[y - 1][x]
                                         + blur_x[y    ][x]
                                         + blur_x[y + 1][x]
```

From the previous algorithm it is possible to observe that the computation is performed at tiles with dimensions of 32×32 pixels. Redundant computation in this approach is only performed for **blur_x** computations that overlap between tiles, parallelism can be performed during the processing of each tile because there is no resource dependency between iterations and locality is achieved as long as the tile region fits in cache.

Since all of these strategies have their drawbacks, it is not straightforward defining which one is the best strategy to use. More precisely, three strategies miserably fail on attending one of the three tradeoffs. The breadth-first fails on locality, full fusion has a lot of redundant computations, and sliding window does not have parallelism. The best strategy could be the tiling, but this is not always the case since it also depends on factors such as image size and hardware capabilities (*i.e.* cache sizes, memory bandwidth and number of available CPU cores and threads).

3.4 COMPILER

The Halide compiler uses the input algorithm and schedule to generate the machine code that implements the Halide pipeline. The compiler does not take any optimization decisions during its execution, it always resorts to the schedule for it. Also, the schedule must not change the program output, it just should affect the organization of the execution. All vectorization instructions, parallelism, data management and GPU kernels are automatically generated by the compiler. Figure 3.2 [58] shows the Halide compilation process.

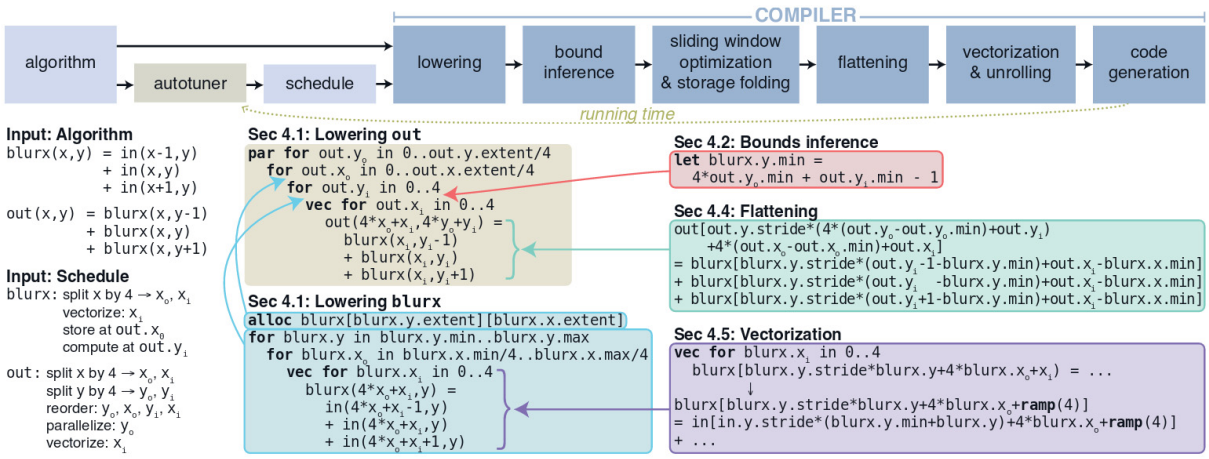


Figure 3.2: Halide compilation process, using an input and the scheduler, the compiler starts lowering the last output function in the pipeline (**out** in this case), then it keeps lowering the previous functions until all the pipeline functions are lowered, generating all of its loops nests based on the defined schedule. Bounds inference is used for calculating the boundaries for each loop and then replacing their symbolic boundaries defined during lowering process. The last steps of the compiler performs low-level optimizations and the code generation for the target by mapping the generated representation into LLVM AST. [58]

Halide compilation is performed throughout several steps, on which the first steps generates the loop nests that represents the algorithm traversal based on the defined schedules, and the last steps are used to perform specific low-level optimizations such as flattening transform for keeping data contiguous in memory, as well as vectorization and unrolling for the target architecture. All the steps are described with more details in the next subsections.

3.4.1 Lowering and Loop Synthesis

First, the compiler performs lowering and loop synthesis. The goal is to provide the loop nests and allocations for the implementation. Starting from the output function, the compiler generates its loop nests and loops for splitting dimensions when necessary – like tiles, for instance –, covering all the required region for the input. It also labels each loop as serial, parallel, unrolled or vectorized. The schedule should define the domain that loops must traverse and their labels.

The compiler then keeps traversing the input functions recursively until all the algorithm functions are lowered. Each producer function must be scheduled to be stored and computed at some loop level – not necessarily the same – of its consumer function. The compiler finds the defined loop levels and injects the buffer allocation and storage right after the loop specified by the store level, and the computation for the producer function right after the loop specified by the compute level. Notice that store levels of a function cannot be inside of its production because functions cannot be stored if not computed yet. After reaching the end of the pipeline, the functions are synthesized into a single set of loops.

3.4.2 Bounds Inference

The implementation generated so far used symbolic boundary variables in the memory allocation and loop definition since no boundary checking was performed yet. The bounds inference step is used for defining these symbolic variables.

Starting from the output function until the input ones, the compiler uses the function definition in the algorithm part to perform interval analysis [43], calculating the expression bounds for the each loop of the function. Dimensions are evaluated one at a time since each dimension corresponds to a different synthesized loop for the function in the generated code. At each function the interval analysis is always done using the previous computed bounds. Finally, bounds inference goes back to the output injecting the definitions for the bound variables generated at the **lowering and loop synthesis** step.

3.4.3 Sliding Window Optimization and Storage Folding

After that, the compiler executes the **sliding window optimization** step by traversing the loop nests of the program. If a function is computed in a more internal loop than its storage with a serial loop separating both operations, then the serial loop can benefit from reusing the computed values in the previous iterations without the risk of race conditions. The Halide compiler uses interval analysis again for shrinking the interval to be computed at each iteration since the region computed by previous iterations does not need to be recomputed.

Another optimization performed in this step is storage folding. If an allocated region is only used inside a serial loop and it is monotonically traversed, then it can be folded by rewriting its access indices, which allows shrinking of the allocated region, reducing peak memory use and working set size [56].

3.4.4 Flattening

Flattening multi-dimensional loads, stores, and allocations is then performed by the compiler, which causes all the buffers and indexes in the implementation to become uni-dimensional. This is done by computing a stride and a minimum for each dimension, and then transforming the accesses of a dimension into the dot product of the accessed index with the computed stride, minus the computed minimum.

The flattening of multi-dimensional operations into its uni-dimensional equivalents reduces the complexity of address generation and enhances cache usage by keeping the data layout contiguous in main memory, which is usually the most appropriate layout for improving cache prediction on modern architectures.

3.4.5 Vectorization and Unrolling

The next step is to perform **vectorization and unrolling** of loops. Unrolling replaces a loop of size n with n sequential statements performing each loop iteration in turn [58], which completely unrolls the loop. Partial unrolling of loops are expressed by transforming one dimension into two and unrolling the inner dimension loop. Vectorization replaces a loop by a single statement. Every reference to the index of the loop is replaced by a special value that represents the vector with the same size as the replaced loop. Type coercion pass is then used to broadcast expressions that contain scalars combined with their respective special values. Finally, Intermediary Representation (IR) nodes are replaced by their vector type relatives, which later generates vectorized code.

3.4.6 Back-end Code Generation

At the last step, **back-end code generation** applies basic optimizations and generates code for the target backend architecture. The compiler first simplifies the IR by performing constant propagation and dead code elimination. Then, it maps the resulting IR to the LLVM IR [33], which at the most part is a one-to-one mapping between both IRs, with some exceptions including the following mentioned by Ragan-Kelley [58]:

- Parallel for loops are lowered to a code which represents its state as a closure data type. The loop body then is lowered to a function call that receives the generated closure as a parameter. This way, it is possible to generate code that enqueues iterations on a task queue to be consumed by threads at runtime.
- Peephole optimization is used on vector codes for using architecture-specific intrinsics, this is performed due to the fact that many vector patterns are difficult to express or LLVM generates poor code for them if passed directly [56].

After mapping to LLVM IR, the LLVM framework takes care of generating the backend machine code using its equivalent IR based on the Halide pipeline.

3.5 CONCLUSION

In this chapter we described the Halide framework, exposed some examples of Halide algorithms and presented its most important primitives to implement algorithms and schedules. We started from explaining the key elements to write Halide algorithms in its functional style, followed by showing schedule primitives for both individual nodes and relations between nodes. Through this, we expect the reader to notice how it is easier to implement and change Halide applications, and how practical and error-free is to experiment and validate different schedules for algorithms on different platforms.

We also explain some examples of schedule based on the presented trade-offs argument proposed by the Halide paper [58]. For this, four different schedules for the Blur algorithm are shown so we can describe their drawbacks and emphasize the importance of managing

the trade-offs among locality, redundant computation and parallelism on image processing applications.

Finally, we show the Halide compiler steps, and detail the process that Halide uses for generating the final code for its backends. The compiler steps also show some important optimizations handled by the compiler such as the flattening of multi-dimensional data operations for a better data layout in favor of performance, and the use of sliding windows and storage folding for reducing redundant computation and peak memory usage.

4 LITERATURE REVIEW

In this chapter we present several work with profiling tools and auto-tuning works that use profiled events in order to achieve better performance variants of an algorithm. In the first part we focus on general purpose programming tools since no works on profiling with domain-specific languages were found. After it, the auto-tuning and auto-scheduling works are used as a motivation to use our new profiler extension. We also expose some Halide auto-schedule attempts and discuss how our profiler can improve these works by providing more performance information.

4.1 PROFILER TOOLS

There is a wide variety of profiler tools that are used for analyzing performance issues and debugging applications. As already mentioned, Linux contains the *perf_events* [15] subset that allows developers to do sampling and instrumenting of running applications in Linux platforms. PAPI [61] is also a library that allows to use performance counters in different types of platforms. We use this section to describe some of these tools and to clarify the reason to use our profiler extension applications rather than directly using one of these tools with Halide.

Gprof [21] is a performance analysis tools for Unix based platforms used for sampling and instrumentation of applications during runtime. Gprof uses the generated profiling code when emitted by the compiler (like when using the **-pg** flags during GCC compilation, for instance) in order to measure and gather the profiling data. It allows to obtain consumed time for each subroutine in the program and to trace the parts of code that are most called during its execution.

Valgrind [48] is a programming tool for profiling and memory debugging. It uses dynamic binary re-compilation, transforming the program into a processor-neutral intermediate representation without executing it. Using this IR, it can perform transformations to add the analysis code and then recompile this transformed version to execute in the host platform. Since the original code does not actually execute in the host platform, Valgrind acts as a Virtual Machine by translating the original program into the instrumented program. Using the injected instrumentation code, one can write Dynamic Binary Analysis (DBA) tools with Valgrind. Default tools present in Valgrind can already check for memory leaks and threading bugs.

Before starting with this research for profiling tools, we also looked for other domain-specific languages that allow to profile their generated programs. We could not find any DSL that generates the appropriate profiling instructions in order to obtain a rich profiling experience. Most domain-specific languages does not take profiling of their generated programs too much into consideration.

Since domain-specific languages usually do not provide profiling of their generated codes, if one wants to profile applications generated by a DSL, it must use general purpose programming languages profilers when available in its host language (if it is an embedded language). This leads to a problem: general purpose language profiling tools will not give informations in the DSL point of view, instead, the results will be shown based on the host language point of view. Nevertheless, using general profiling tools such as the available ones will probably not give detailed outputs when dealing with DSL generated applications. If we would take Halide, for instance, we cannot profile an application's specific functions individually using the profiler tools that can be used with C++ developed applications, because Halide functions are just objects of the `Func` class in C++ point of view.

4.2 AUTO-TUNING STRATEGIES USING PROFILED EVENTS

In order to give some motivation of how our work can contribute on generating code with better performance, we use this section to expose some examples on auto-tuning and auto-scheduling using online profiling data. Notice that even simple machine learning driven automatic systems for auto-tuning or auto-scheduling use profiled parameters such as execution time of the application. Feeding these systems with the right profiled data in the critical performance regions of each variant will turn it to converge faster and generate better code.

Besides this, one may not restricts itself to reach just time-efficient generated code, it can also worry on generating better energy-efficient applications. Since for this it is necessary to evaluate energy costs for each variant in order to select the best ones during the learning steps, this is only possible using a profiler tool. This type of optimization can be even more important at the present moment considering the demand for image processing algorithms on resource-limited environments such as mobile devices that depends on the energy stored in batteries, for instance.

Also, when dealing with manual tuning of applications, profilers and code optimization walk together. It is not uncommon to deal with bad performance code for reasons that are not too intuitive to identify. Using information provided by profilers can always be helpful to understand why a specific code has bad performance.

Work on tuning GPGPU applications already relies on profiling to find the best kernel configurations. Guerreiro et al [23] select the best GPU operation frequency, grid and block sizes configurations aiming performance and energy-aware optimization. The auto-tuner decisions are based on measurements of time and variation in energy consumption. However this work does not look for better algorithm alternatives and just takes into consideration the kernel configurations and GPU operation frequency.

Weber et al [71, 72, 73] uses guided profiling integrated with an auto-tuner for finding the most suitable array layout on GPUs and optimize the array accesses in CUDA applications. The work uses a predictor to reduce the time required for empirical profiling, and a watch-dog that also reduces the profiling time by terminating the execution of the kernel in bad configurations.

Dotsenko et al [16] present an auto-tuning framework to solve fast fourier transformation kernels on GPUs using profile-based techniques. The work develops a FFT kernel generator that produces different kernel variants based on an extensive analysis of factors that affects performance on FFT kernels. Also, it prunes the search space by using a pruning heuristics that discards kernel variations that exceeds shared memory or thread count limits, as well as variants that exceed the register limit by a small threshold. The work measures runtime of a generated column kernel for each legal sample stride, and records time per element in the performance database. If the number of slow samples take more than a specific percentage more time to run than the best recorded time, do not profile the kernel further. In general, it delivers performance 3× higher than hand-tuned FFT kernels.

Besides GPU auto-tuning using online-profiling, CPU works are also present. Chen et al [9, 8, 10] optimizes *work-stealing* [4] policies scheduling on NUMA-based memory architectures by using online profiling and auto-tuning on multisocket multicore CPUs. The work provides a LAWS (Locality-Aware Work Stealing) scheduler that aims to reduce the number of remote memory accesses (accesses to memory nodes that are bounded to a different socket than the local one) in a shared-cache-friendly way. Online-collected information and auto-tuning is used for optimizing the shared cache usage. Results show that the work improves the performance of memory-intensive applications up to 54.2% on AMD-based experimental platforms and up to 48.6% on Intel-based experimental platforms compared with traditional work-stealing schedulers.

More close to our work, Halide [57, 58, 56] already has a simple profiler that can estimate memory footprint, time, and average number of threads running in parallel for each function in the pipeline. These estimates are also used for decision making on automatic generation of schedules [45]. However, there are few available measurements and they do not help much when dealing with larger and more complex pipelines, because most of these measurements can be considerably influenced by others. For instance, parallelism can be influenced by memory bandwidth if there is no bandwidth available at a certain moment, as threads will be stuck waiting for transferring data into the memory. Time also does not express too much since it is usually influenced by several other factors, i.e., any resource bottleneck can significantly increase time. These influences among measurements can end up misleading the schedule developer or auto-tuner and prevent it to find the true bottlenecks for improving the schedule.

Pecenin [53] uses reinforcement learning in order to automatic generate better schedules for Halide algorithms. The work maps the scheduling problem as a Markov decision process and integrates it into a neural-networks based agent that adds approximation functions for large spaces, called Proximal Policy Optimization (PPO). At the moment it just uses time to evaluate schedules and therefore it can really benefit from our profiler extension by feeding the learning agent with more performance-critic informations.

4.3 CONCLUSION

In the first part of this chapter, we introduced some profiling tools for general purpose applications and discussed why it is interesting to have a profiler tool for a domain-specific language code. Our point is that we want to measure performance events in the DSL generated code in a more fine-grain level, and that is not possible if the DSL compiler does not generate the proper annotated code.

Besides this, we also expose some works that use online profiling data for automatic optimization or scheduling. These works are shown as a motivation to use our tool for generating better schedules for Halide applications. Also, as showed, there are already some attempts to auto-schedule Halide algorithms, we intend to improve results for this area with our profiler extension.

5 HALIDE DSL PROFILING EXTENSION

In this work [37] we present an extension to Halide DSL that includes the appropriate markers for instrumenting the generated code. Figure 5.1 gives an overview of how our extension works in both Halide compilation and runtime steps. The provided functions allow us to include the markers in the generated code (depicted by the `profile_enter_func()` and `profile_leave_func()` calls in the generated code in Figure 5.1). These markers then triggers the runtime code that end up calling the profiling library to start and stop counting the performance events.

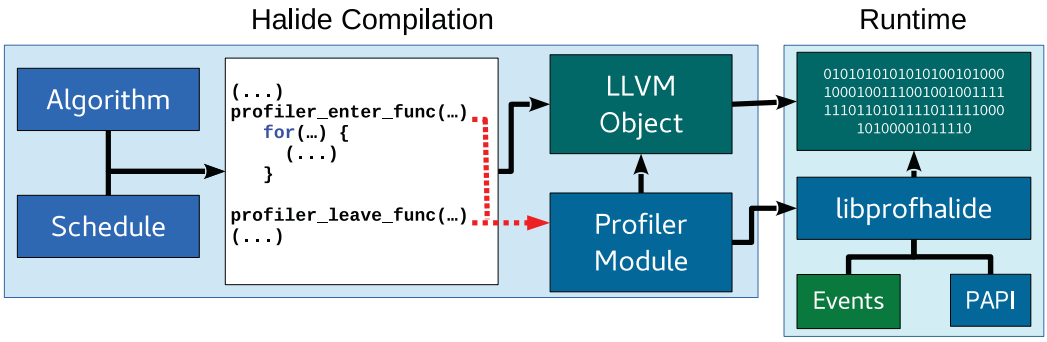


Figure 5.1: Halide code generation with profiling scheme.

5.1 LANGUAGE EXTENSIONS

The extension consists of two member functions to the `Func` class named `profile()` and `profile_in()` and a third function named `profile_at()`. They can be used directly in the Halide schedule to define where the code should be profiled and how:

`Func::profile()`: injects the profiler markers at one of the stages (production, consumption or all) of the Func.

`Func::profile_in()`: similar to `Func::profile()`, but only affects the stages inside a parent function. Useful for turning the profiler off for specific parts of the code inside a region;

`profile_at()`: injects the profiler markers at the loop level of the Halide code specified at the `Func` parameter.

Listing 5.1 shows the C++ headers for the new functions our extension implements.

Listing 5.1: Headers of the profiling functions defined by the proposed Halide extension.

```
class Func {
  ...
  /** Profile this function. */
  Func &profile(
    int stage = PROFILE_PRODUCTION, bool show_threads = false,
    bool enable = true, int granularity = 1 );

  /** Profile this function stages inside
     a parent function's production stage. */
  Func &profile_in(
    Internal::Function &parent,
    int stage = PROFILE_PRODUCTION, bool show_threads = false,
    bool enable = true, int granularity = 1 );
  ...
};

/* Profile a loop level defined by a function (Func) and a
   dimension (Var) */
void profile_at(
  Func f, Var var, bool show_threads = false,
  granularity = 1);
```

These extensions functions have following parameters:

`stage`: since Halide functions are converted into producer-consumer relations in the generated code, this parameter defines whether to profile the function production stage (`PROFILE_PRODUCTION`), the function consumption stage (`PROFILE_CONSUMPTION`) or both production and consumption stages (`PROFILE_ALL`). The default value for this parameter is `PROFILE_PRODUCTION`;

`show_threads`: allows to report the results for each individual threads in this region (`true`), as opposed to the default behavior of just reporting the summation of the counters for all threads (`false`);

`enable`: allows to enable/disable profiling, which is useful because profiling markers are enabled recursively in a loop nest and sometimes we need to disable profiling for specific regions inside this loop. Disabling profiling for specific regions can be beneficial if we are not interested in counting events inside them because no time is spent triggering the profiler counters on disabled regions;

`granularity`: is a parameter that defines the number of loop iterations between measurements. The performance events are only counted in a fraction of the loop iterations, thus reducing the impact of profiling in the overall perfomance of the code;

`parent`: is a parameter of the `profile_in()` function to define in a more specific way at which stage (or `Func`) we want our changes (enabling or disabling the profiler) to take effect.

In order to exemplify the proposed Halide profiling extension as well as the modifications it produces in the generated code, we use the image *Blur* code with tiling schedule introduced in Listing 3.1. In this algorithm we have the **blur_x** and **blur_y** functions, and in the tiling schedule the **blur_y** function consumes values produced by **blur_x**. Production and consumption of the

blur_x values occur in levels of tiles inside the production of **blur_y**, as can be seen in Listing 5.2:

Listing 5.2: Structure of the code generated by Halide for the *Blur* algorithm with a tiling schedule of 32×32 pixels.

```
1  produce blur_y {
2    ... // (1) Warm up code for blur_y production
3    tile 32x32 { // Go through the domain in tiles of 32x32
4      produce blur_x {
5        ... // (2) Produce blur_x values using the input
6      }
7      consume blur_x {
8        ... // (3) Consume blur_x values to produce blur_y
9      }
10   }
11   ... // (1) Warm up code for blur_y production
12 }
```

If we want to profile the production of **blur_y** in this example, our profiler must include markers separating all the regions inside the production of **blur_y**. In this example we have three regions:

1. the warm-up code (Listing 5.2, lines 2 and 11) composed of all code not inside a production or consumption region;
2. production of **blur_x** (Listing 5.2, lines 4-6); and
3. consumption of **blur_x** (Listing 5.2, lines 7-9).

With the proposed extension this can be achieved by adding the following call to the `profile()` function inside the Halide code, *i.e.* before the `return` command (line 14) in Listing 3.1:

```
blur_y.profile(PROFILE_PRODUCTION, false, true);
```

The profiler recursively includes the necessary markers for the different regions in the generated code of the **blur_y** loop nest, enabling profiling in all regions related to the production of **blur_y**. Notice the first parameter `PROFILE_PRODUCTION` tells the profiler we want to profile the production stage of the **blur_y** function, the second parameter `false` tells we do not want to show each thread separately, which is only useful in parallel regions, and the last parameter `true` tells we want to enable the profiler.

Now, suppose we still want to profile the production of **blur_y** values, but we do not want to profile the production of **blur_x** values inside of it. We can achieve this by adding following line of code:

```
blur_x.profile_in( blur_y, PROFILE_PRODUCTION,
                   false, false );
```

The first parameter tells the profiler that this call should just affect child relations of the **blur_y** production loop nest, so that if there are other **blur_x** consumption regions inside other functions, they would not be affected. The remaining parameters are similar to the `profile()` call.

We list the relevant markers created by the proposed profiling extension and provide a brief explanation of their functionality:

`pipeline_start/stop():` define the start and end of the pipeline execution, initializing all performance counters and the profiler API (**`start`**), and reporting the results (**`stop`**);

`enter/leave_warmup_region(f):` start and end markers for counting the performance events in the warm-up region of a function **`f`**;

`enter/leave_func(f, prod):` start and end markers for a production or consumption region of the function. **`f`** defines the function to be profiled and the **`prod`** parameter is a zero or one multiplexer that indicates if the counters should be stored in the production or in the consumption counters;

`enter/leave_loop(l):` start and end markers for a loop level region. **`l`** defines the region for counting the events and store the values in the loop level counters;

`enter/leave_parallel_region():` defines a parallel region of the code, used to manage thread structures that traces the threads individually; and

`incr/decr_active_threads():` defines the region where a thread is active, used to manage the event set needed to count events for the thread.

In the *Blur* algorithm we profile in this work, the generated code of Listing 5.2 is modified by the proposed Halide profiling extension injecting the region markers, generating the code shown in Listing 5.3. Notice the markers that define the start and end of the warm-up region (lines 4-6 and 18-21) as well as the production (lines 9-11) and consumption (lines 14-16) regions of the Halide functions. Since this schedule does not use parallelism, there is just one active thread during the entire execution of the pipeline, and only two markers at the start and end of the code are necessary to start and stop the event sets (lines 1-2 and 23-24).

Listing 5.3: Instrumented code for the *Blur* algorithm with tiling schedule of 32x32 pixels.

```
1  halide_papi_pipeline_start()
2  halide_papi_incr_active_threads()
3  produce blur_y {
4    halide_papi_enter_warmup_region(blur_y);
5    ... // Warm up code for blur_y production
6    tile 32x32 { // Go through the domain in tiles of 32x32
7      halide_papi_leave_warmup_region(blur_y);
8      produce blur_x {
9        halide_papi_enter_func(blur_y, true);
10       ... // We produce the blur_x values using the input
11       halide_papi_leave_func(blur_y, true);
12     }
13     consume blur_x {
14       halide_papi_enter_func(blur_x, false);
15       ... // Use produced blur_x values to produce blur_y
16       halide_papi_leave_func(blur_x, false);
17     }
18     halide_papi_enter_warmup_region(blur_y);
19   }
20   ... // Warm up code for blur_y production
21   halide_papi_leave_warmup_region(blur_y);
22 }
23 halide_papi_decr_active_threads()
24 halide_papi_pipeline_stop()
```

In order to inject these markers, we traverse the Abstract Syntax Tree (AST) of the pipeline and perform modifications when finding producer-consumer relations or loop levels that

should be profiled with our functions (`profile()`, `profile_in()` and `profile_at()`). When finding a producer-consumer relation node that must be instrumented, we look deeper in the AST to check if this node contains child producer-consumer relations. If that is the case, we inject the `enter_warmup_region()` marker as the first part of the current node body, and the `leave_warmup_region()` as the last part of the current node body. We then traverse all internal producer-consumer nodes inserting the `leave_warmup_region()` before entering them (Listing 5.3, line 7) and the `enter_warmup_region()` after leaving (Listing 5.3, line 18). This procedure is executed recursively.

When producer-consumer relations nodes that have no producer-consumer child nodes are found, we insert the `enter_func()` and `leave_func()` markers in the first and last parts of the body, respectively. For loops, the procedure is analogous to the nodes without children, but using the loop markers instead.

We simplify the tree by removing unnecessary `enter_warmup_ region()` calls immediately followed by `leave_warmup_region()` calls (in Listing 5.3, markers between the production and consumption of **blur_x** on lines 12-13 were removed).

Markers `pipeline_start` and `pipeline_end` are inserted at the start and end of the AST, and markers for the activation and deactivation of a thread expressing the execution of the serial region (Listing 5.3, lines 2 and 23). If a parallel loop is found, we insert the parallel region markers and the markers to control the activation and deactivation of threads.

5.2 PROFILER RUNTIME CODE

So far we explained how to include calls to the profiler markers into the code generated by the Halide compiler. Figure 5.1 illustrates how the profiling code (white box) is called during the Halide application run-time. To control the profiling for all the different regions in the Halide code and also access the hardware performance counters, we need to implement these markers in a Halide run-time module (because all functions executed during runtime must be implemented in a runtime module). The Halide run-time module *Profiler Module* (depicted in Figure 5.1) contains counters for each of the profiled functions' production, consumption and warm-up regions and/or standalone profiled loops.

However, in order for the run-time code to access hardware performance counters it needs platform-specific code and this cannot be implemented in the *Profiler Module* (Halide modules must be platform-agnostic). A separate library, the **libprofhalide**, is used to overcome this restriction, and is responsible for calling the platform-specific functions. In our case, **libprofhalide** uses the performance counters provided by the **PAPI** library [61], but any profiling tool such as perf_events [15] or Likwid [66] could be used instead. The available performance events for profiling are defined by these platform-specific libraries and the processor capabilities, not being limited in any way by the proposed Halide extension. The events to be measured (i.e. L1_CACHE, FLOP) are defined in a configuration file loaded by **libprofhalide** (**Events** box in Fig. 5.1).

6 EXPERIMENTAL RESULTS

In order to demonstrate the use of the proposed Halide profiling extension we chose to use the four schedules for the *Blur* algorithm [58] depicted in Figure 3.1: *breadth first*, *full fusion*, *sliding window* and *tile 32x32*. The aim is to compare their theoretical strengths and weaknesses (trade-off between data locality, redundant computation and parallelism) with the real executions of these schedules on a general purpose processor. The Blur code for the algorithm and schedules used for our experiments can be seen in Appendix A.1 and Appendix A.2, respectively.

Also, we use four different schedules for the *Interpolation* algorithm. The *Interpolation* performs multi-scale interpolation using an image pyramid to interpolate pixel data for seamless computing. The resulting pyramids are chains of stages that resample the image over small stencils. We use 5 levels of resampling, and so the stages can be defined as follows:

- **downsampled[0-4]**: Stages that compute the smaller samples of the image, where downsampled[1] computes the smaller version for the downsampled[0] output, for instance. Hence, downsampled[1] is a consumer function of downsampled[0].
- **interpolated[0-4]**: Stages that use interpolation to compute bigger samples of the image, where interpolate[0] computes the bigger version of interpolate[1]. Hence, interpolate[0] is a consumer function of interpolate[1].
- **normalize**: Stage that performs the normalization using the alpha channel value of the image. This stage simply divides the values in the RGB values of a pixel by the value in the alpha channel.

There are other **Func** objects defined in the *Interpolation* algorithm, but they are fully-inlined in all of our schedules, and then just represents additional computation in the generated code. We used the following four schedules in our experiments for the *Interpolation* algorithm:

- **flat**: Similar to breath-first in the Blur algorithm, this just computes all the stages at root level. Other schedules are derived from this one.
- **flat_vec**: Same as **flat**, but this time performing vectorization in the x axis of all stages. The **normalize** stage does not perform vectorization.
- **flat_par_vec**: Same as **flat_vec**, but this time also computing the stages in parallel. On interpolated[0-4] stages, also unrolls the loops at x and y axis by a factor of two.
- **flat_vec_sometimes**: Same as **flat**, performing vectorization in the x at levels that compute bigger samples. Since we use just 5 levels, we just use vectorization in the first levels (**downsampled[0]** and **interpolated[0]**). The **normalize** stage does not perform vectorization.

The Interpolation code for the algorithm and schedules used for our experiments can be seen in Appendix B.1 and Appendix B.2, respectively.

6.1 PROTOCOL

We measure time, number of L1 cache misses, number of float instructions and L3 cache data transfer volume for each schedule, with single thread and four threads, on 8 Megapixel (3840x2160) and 44 Megapixel (10240x4320) images. These image sizes were chosen for their common use on high-definition devices (4K and 10K resolution) and therefore their relevance for real-world image processing applications.

The experiments were executed on an Intel Core i5-7500 CPU at 3.40GHz, with 4 cores per socket and 1 thread per core (no simultaneous multi-threading), L1, L2 and L3 caches of sizes 32kB, 256kB and 6MB respectively, and 8GB of RAM. For the parallel schedules we used 4 threads, pinning them to the same core so to enforce the use of the same L1 and L2 caches, improving data locality.

We performed 30 executions of each schedule for each PAPI event on each image size. For the *Blur* algorithm, we present the schedules with one and four threads (except *sliding_window*), and afterwards we present the results for the same schedules (both serial and parallel) with vectorization. For all tests, we present the averaged results of all the iterations. We are aware that the average results do not show the variability of the process. We previously computed the standard deviation for each counter on all the 30 executions and noticed that the variability is very small for each counter (for FLOP there is no variation, for the remaining events the variation is about 2%).

6.1.1 Blur

Measurements were made separately for **blur_x** and **blur_y** whenever there were separate production/consumption regions. Following code was inserted after the *breadth first* schedule:

```
blur_x.profile( PROFILE_PRODUCTION );
blur_y.profile( PROFILE_PRODUCTION );
```

For *full fusion* and *tile 32x32* schedules only **blur_y** has to be profiled because **blur_x** is produced inside it:

```
blur_y.profile( PROFILE_PRODUCTION );
```

On the *sliding window* schedule both productions of **blur_x** and **blur_y** are performed inside the most inner loop, and profiling these stages individually greatly increases the time overhead for calling the profiling functions. Therefore this schedule was profiled differently, at **blur_y**'s outer loop, not allowing to separate the results from **blur_x** and **blur_y** counters:

```
profile_at( blur_y, c );
```

Parallel versions of the schedules were profiled by individual threads. No parallel version for the *sliding window* schedule was tested due a potential race condition in its Halide implementation.

6.1.2 Interpolate

Listing 6.1: Code for profiling all the Interpolation stages.

```
1 for(int l = 0; l < levels; ++l) {
2   downsampled[l].profile(PROFILE_PRODUCTION);
3   interpolated[l].profile(PROFILE_PRODUCTION);
4   normalize.profile(PROFILE_PRODUCTION);
5 }
```

Listing 6.1 shows the snippet of code used to profile all the stages in the *Interpolation* algorithm. Notice the simplicity of just iterating over each function and using the `profile` function for each stage. With this code we can obtain the events for all the different parts of the *Interpolation* pipeline. The parallel schedule **flat_par_vec** is also profiled by individual threads in our results.

Although we profile all the events separately, we show the stages as merged in the result figures to not pollute our charts. For instance, instead of showing the **downsampled[0]**, **downsampled[1]** ... **downsampled[4]** stages separately, we show a single stage **downsampled[0-4]** representing all these stages.

6.1.3 Events Measured

Following PAPI events were measured:

`PAPI_L1_DCM`: PAPI preset for counting L1 data cache misses. Poor locality schedules tend to increase the number for this events;

`PAPI_SP_OPS`: PAPI preset for counting the number of single precision floating-point instructions. Schedules that performs lots of redundant computation tend to increase the number for these events;

`PAPI_VEC_SP`: PAPI preset for counting the number of single precision vector instructions. This event is used when counting schedules that perform vectorization. This event can be used instead of **PAPI_SP_OPS** when dealing with vector instructions.

`L3_DATA_VOLUME`: a composed event to count Bytes transferred to/from L3 cache, which is the first level of cache memory and is shared among the CPU cores. This event allows to evaluate schedules' utilization of the (slow) main memory bus. Increase in data volume indicate redundant load/store instructions. It is computed as:

```
L3_DATA_VOLUME = CACHE_LINE_SIZE *
          (L2_LINES_OUT:NON_SILENT + L2_RQSTS:MISS)
```

where:

`L2_LINES_OUT:NON_SILENT`: CPU native event that counts the number of L2 cache lines evicted;

`L2_RQSTS:MISS`: CPU native event that counts the number of requests that miss the L2 cache.

6.2 RESULTS

6.2.1 Blur

On Figures 6.1 and 6.2 we can observe that *full fusion* has a slightly lower execution time than the *tile* schedule for both image sizes, and they are the best performing schedules. The worst performing schedule is the *sliding window*. Parallel efficiency is not very good on any schedule (less than 50%), indicating this is a memory bound problem and simply increasing the number of processors might not be the best strategy.

Also, it is important to analyze the impact of using vectorization. Vectorization is performed to increase the Instruction Level Parallelism (ILP) of the CPU. Hence, using

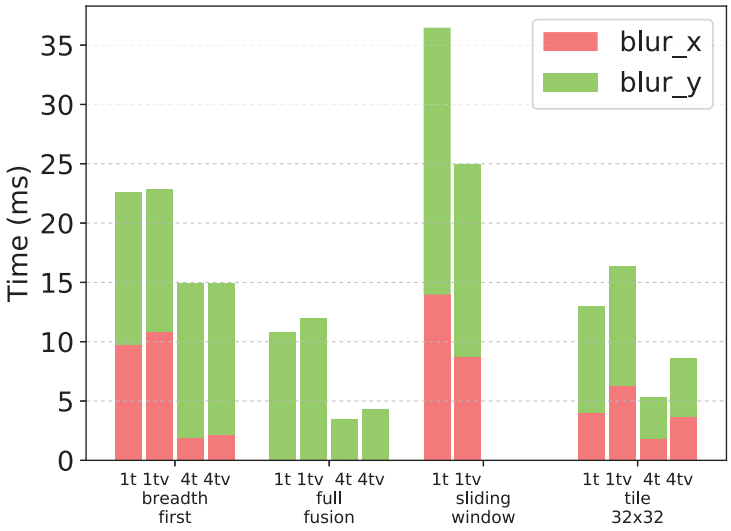


Figure 6.1: Execution time in ms for each Blur schedule, with single thread (1t), 1 thread with vectorization (1tv), four threads (4t) and four threads with vectorization (4tv), on 8 Megapixel images.

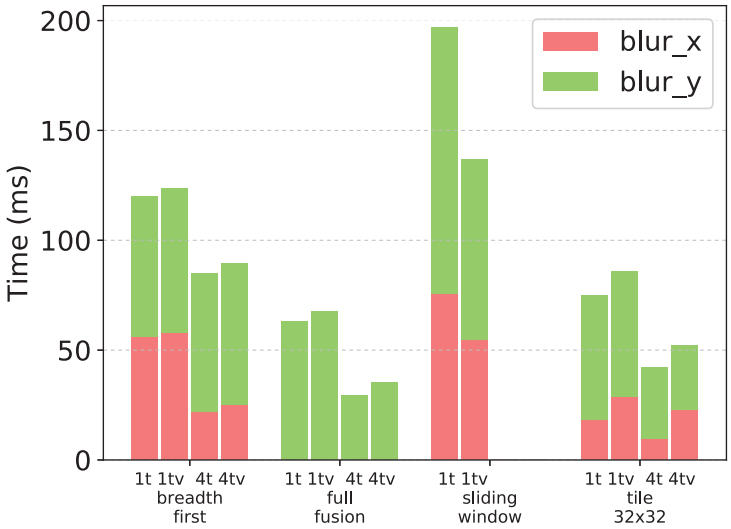


Figure 6.2: Execution time in ms for each Blur schedule, with single thread (1t), 1 thread with vectorization (1tv), four threads (4t) and four threads with vectorization (4tv), on 44 Megapixel images.

vectorized code tends to increase the processor throughput. Notice that there is no relevant performance improvement on using vector instructions in the majority of the schedules. This is a possible indication that the Blur algorithm is limited by the memory bandwidth. To perform the computations, it is necessary to wait for the data to be loaded from the memory. The computation throughput is limited by this wait, and because of this, vectorization may not be an efficient choice here. Also, some schedules perform a little worse when using vectorization, which could be explained due to the overhead provided on using vector instructions.

Figures 6.3 and 6.4 show that the *full fusion* schedule presents the smallest number of cache misses, since it has better data locality (each pixel is accessed only once). The *breadth first* schedule shows the biggest number of L1 cache misses and, as expected, they are higher in the **blur_y** function because there the computation traverses the image in a strided memory access pattern, resulting in bad data locality. The results are very similar to the schedules with and without vectorization, which is expected since vectorization should not affect the number of L1 cache misses.

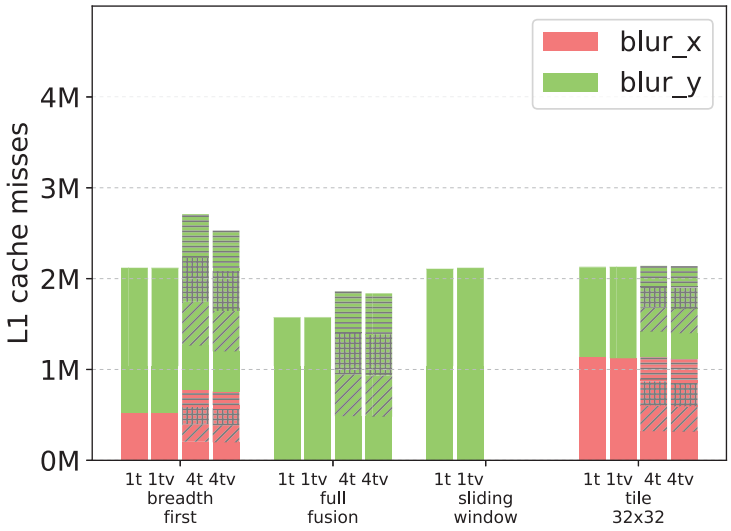


Figure 6.3: Number of L1 cache misses for each Blur schedule, with single thread (1t), 1 thread with vectorization (1tv), four threads (4t) and four threads with vectorization (4tv), on 8 Megapixel images.

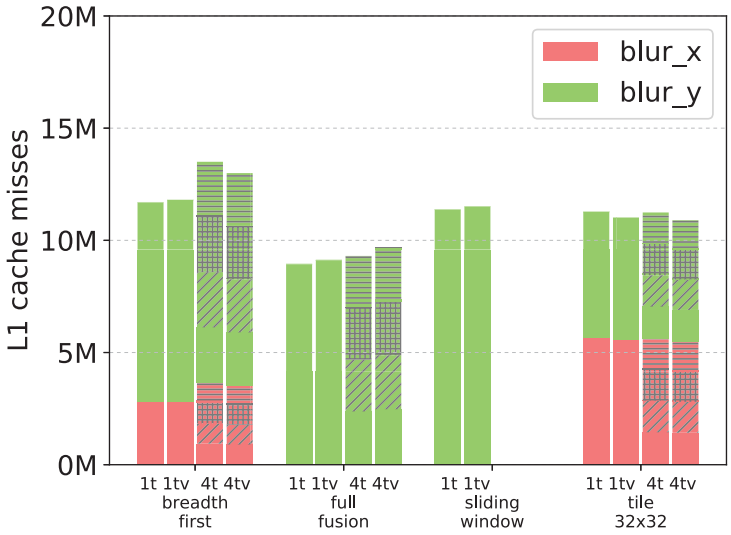


Figure 6.4: Number of L1 cache misses for each Blur schedule, with single thread (1t), 1 thread with vectorization (1tv), four threads (4t) and four threads with vectorization (4tv), on 44 Megapixel images.

Redundant computation can be analyzed through the number of floating-point instructions (FLOP) in Figures 6.5 and 6.6. The number of FLOP is similar for all schedules except for *full fusion*, which is reasonable since full-fusion does not use a buffer to store intermediate results of the **blur_x** computation, and computes the blurring in the x-direction three times for each pixel. Since we are using vectorization with 4-width vectors, it is possible to notice that the number of floating-point instructions in the schedules with vectorization is about 1/4th the number of instructions in the schedules without vectorization, because each vector instruction performs 4 operations at a time.

As mentioned before, vectorization is not an efficient choice for this algorithm since it presents a memory bound situation. Figures 6.7 and 6.8 shows the number of floating-point operations per milliseconds (MFLOP/ms), which is a good parameter to measure the CPU effectiveness. We can notice that for practically all schedules (except *sliding window*), their versions with vectorization present a smaller efficiency, therefore no benefit is achieved with vectorization. Also observe the number of operations per second is significantly lower in the

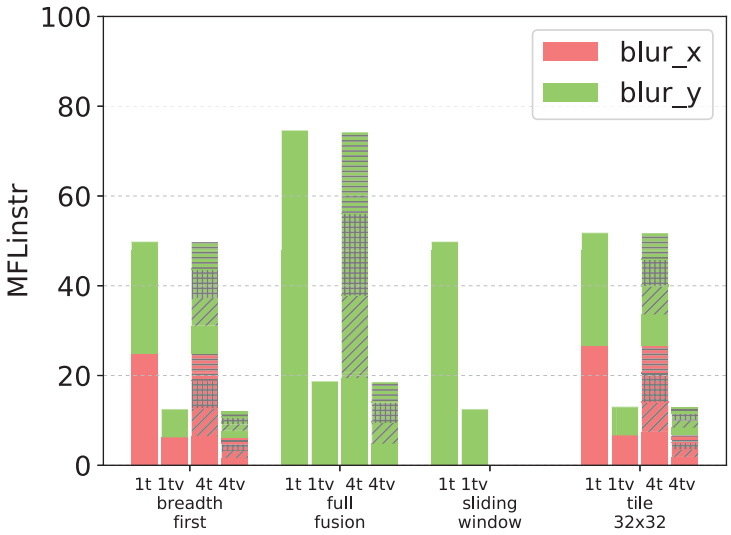


Figure 6.5: Number of float instructions for each Blur schedule, with single thread (1t), 1 thread with vectorization (1tv), four threads (4t) and four threads with vectorization (4tv), on 8 Megapixel images.

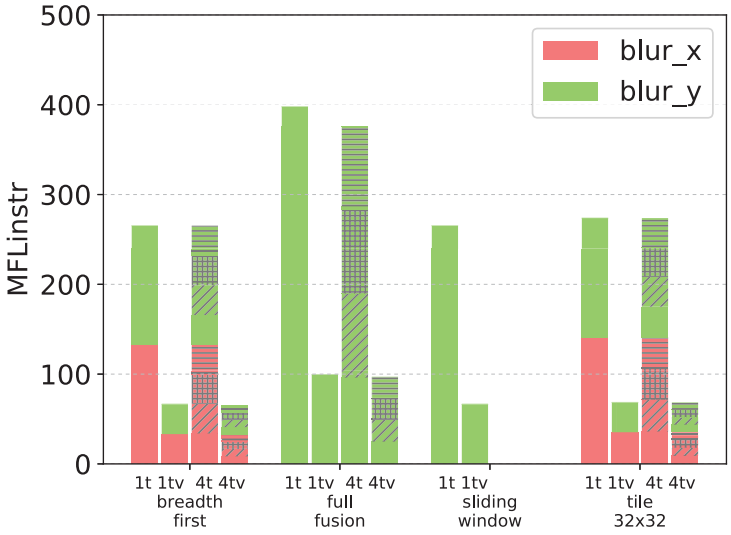


Figure 6.6: Number of float instructions for each Blur schedule, with single thread (1t), 1 thread with vectorization (1tv), four threads (4t) and four threads with vectorization (4tv), on 44 Megapixel images.

blur_y stage. As we mentioned when analyzing the number of cache misses, this happens because this stage performs strided access in the memory which incurs in significantly worse locality compared to the **blur_x** stage.

Data volume measured at the L3 cache level can be seen in Figures 6.9 and 6.10. We notice that the parallel versions of the *breadth first* and *full fusion* schedules transfer more data than their single threaded counterparts. *Full fusion* transfers more than twice as much bytes in the 44 Megapixels image because threads split the image in y-direction, one line per thread, and thus each thread has to access the same memory as its neighbours. *Breadth first* accesses almost three times as much memory on the **blur_y** function for the same reason. This also explains their bad parallel performance: although they realize the same amount of FLOP in total, threads make redundant memory access competing for the memory bus.

None of the measured events is capable of explaining the *sliding window* performance. It has average cache misses and float instructions, and low data volume, but is much slower than all other schedules. In order to explain this it is necessary to hypothesize on the reasons and

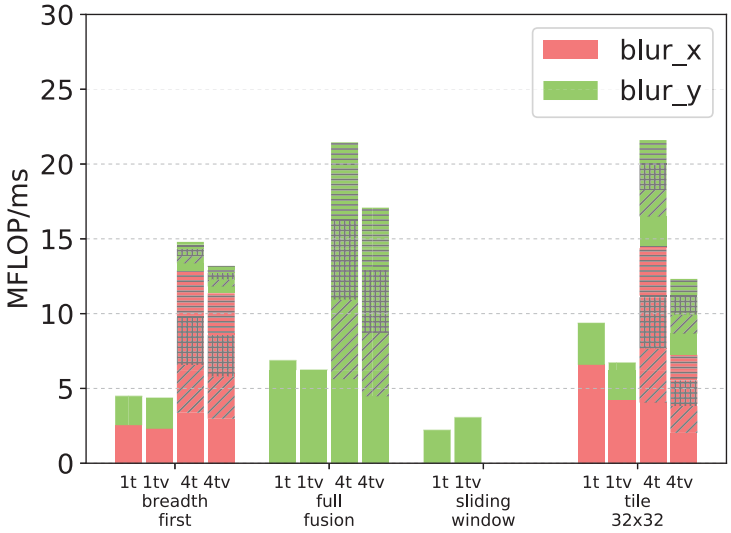


Figure 6.7: Number of float operations per second for each Blur schedule, with single thread (1t), 1 thread with vectorization (1tv), four threads (4t) and four threads with vectorization (4tv), on 8 Megapixel images.

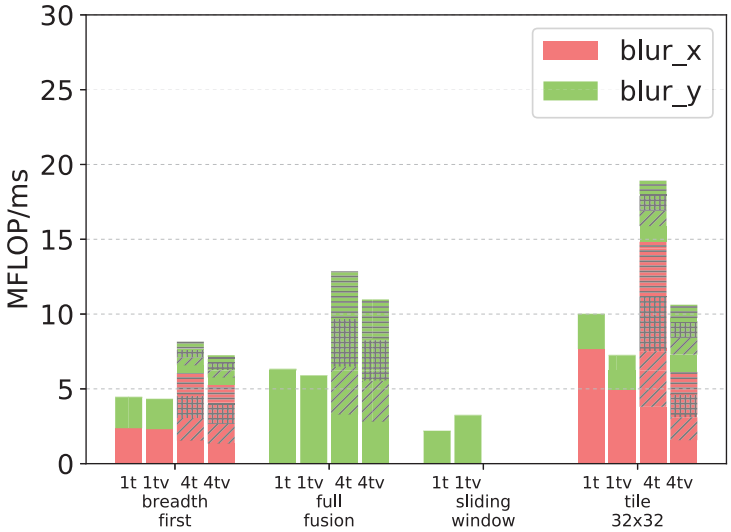


Figure 6.8: Number of float operations per second for each Blur schedule, with single thread (1t), 1 thread with vectorization (1tv), four threads (4t) and four threads with vectorization (4tv), on 44 Megapixel images.

measure events that confirm the hypothesis. One possibility is that *sliding window* is stalled because of excessive function calls in the inner-most loop. This could be verified by counting number of instructions executed per second. Also, since we cannot identify the bottleneck in this schedule, we can at least deduce from these results that this bottleneck does not harm the use of vectorization instructions, because its the only schedule that benefits from the use o vectorization when looking at the execution time results in Figures 6.1 and 6.2.

For the presented schedules we can conclude that optimizing locality is much more effective than redundant computations for this specific algorithm. Using more computational intensive stages in a pipeline may show the opposite behavior and that is something that can be identified using a profiler.

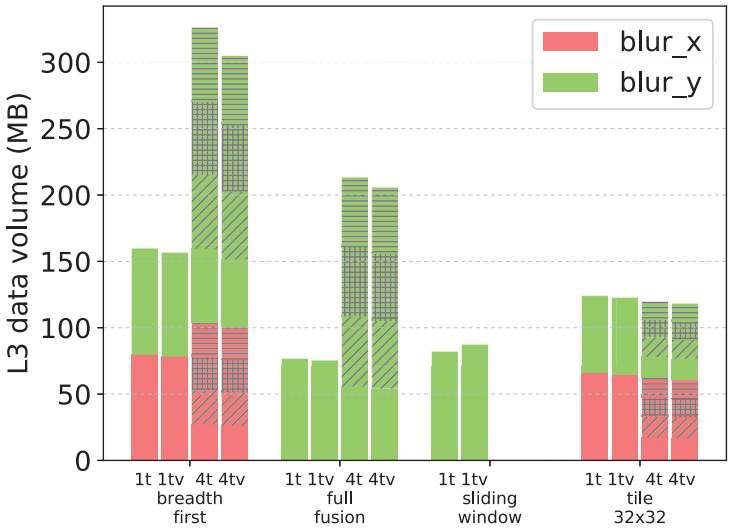


Figure 6.9: L3 data volume for each Blur schedule, with single thread (1t), 1 thread with vectorization (1tv), four threads (4t) and four threads with vectorization (4tv), on 8 Megapixel images.

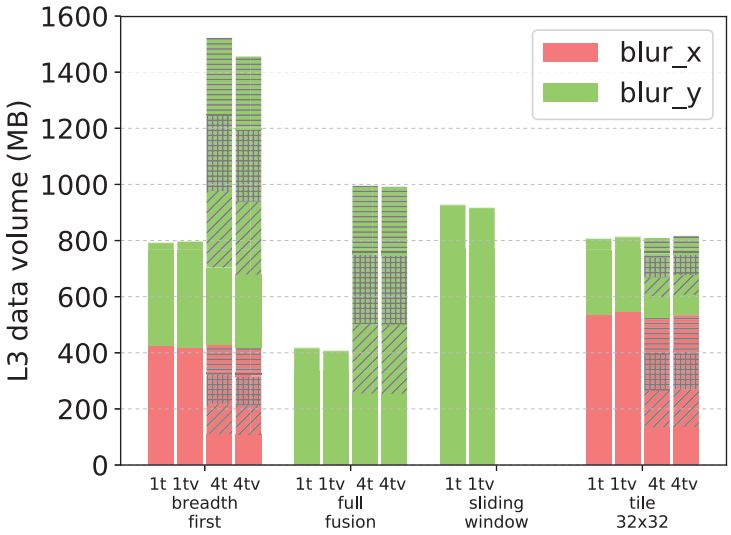
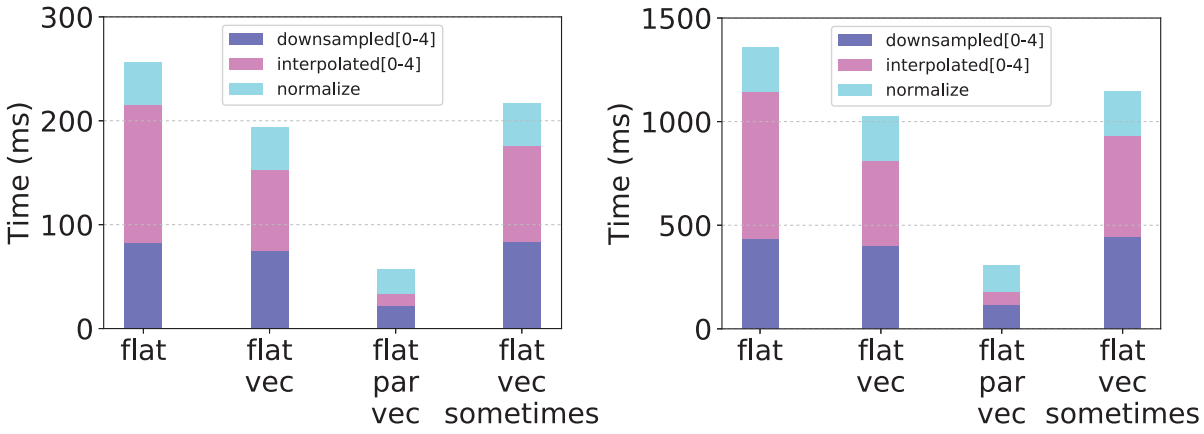


Figure 6.10: L3 data volume for each Blur schedule, with single thread (1t), 1 thread with vectorization (1tv), four threads (4t) and four threads with vectorization (4tv), on 44 Megapixel images.

6.2.2 Interpolation

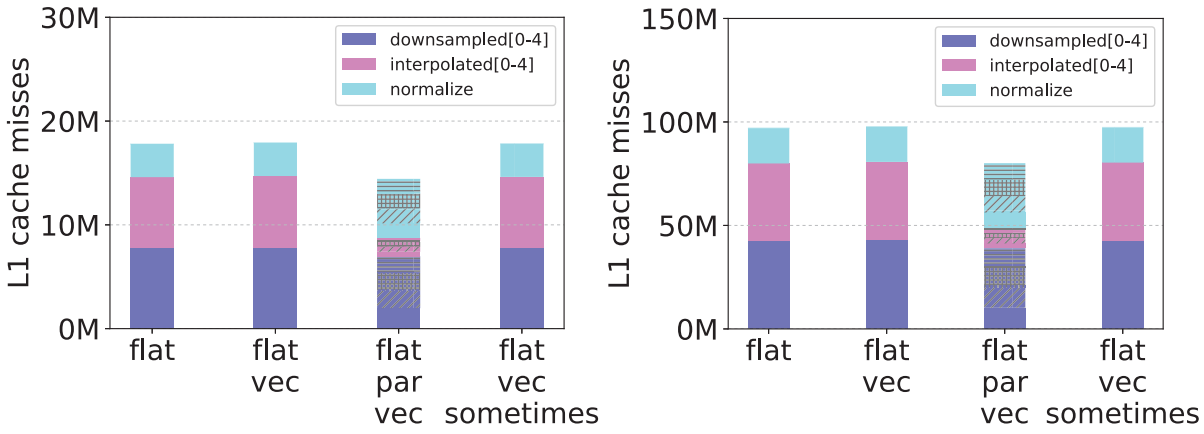
On Figures 6.11(a) and 6.11(b) we see the time results for the interpolation schedules with 8 Megapixel and 44 Megapixel image sizes, respectively. Notice that in this case, differently from the Blur, adding vectorization to the schedules actually improves performance. This happens because interpolation stages are more computational intensive, and because of it, they benefit from increasing the processor throughput.

Figures 6.12(a) and 6.12(b) show the number of L1 cache misses for each schedule with 8 Megapixel and 44 Megapixel image sizes, respectively. It is possible to notice that the number of cache misses is similar for all schedules, except for the **interpolate** stages in the **flat_par_vec** schedule. This probably happens because in this schedule both the **x** and **y** axis are unrolled by a factor of two. Unrolling can help to reduce the number o cache misses when we have strided accesses in the image. Without unrolling, each iteration will load and entire cache line from the



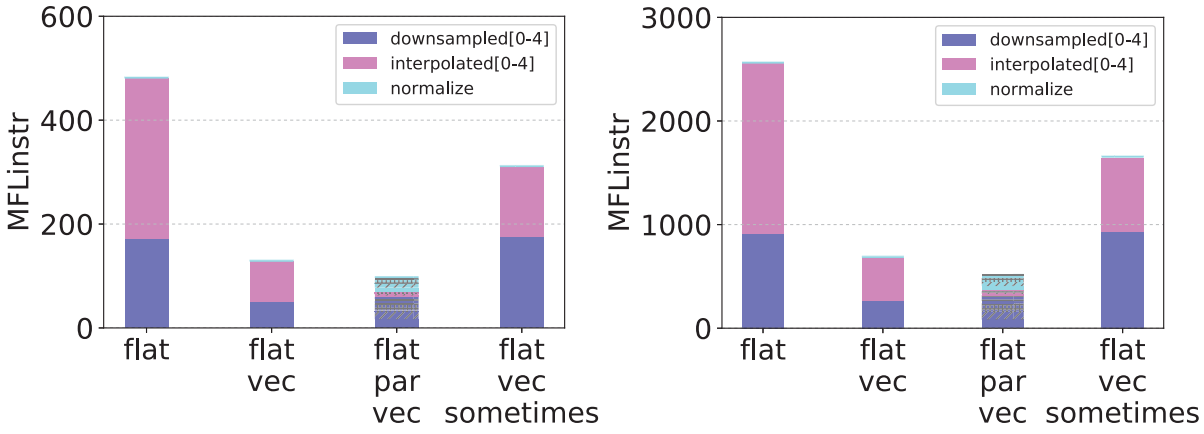
(a) 8 Megapixels (b) 44 Megapixels

Figure 6.11: Execution time in ms for each Interpolate schedule on 8 Megapixels and 44 Megapixel images.



(a) 8 Megapixels (b) 44 Megapixels

Figure 6.12: Number of L1 cache misses for each Interpolate schedule on 8 Megapixel and 44 Megapixel images.

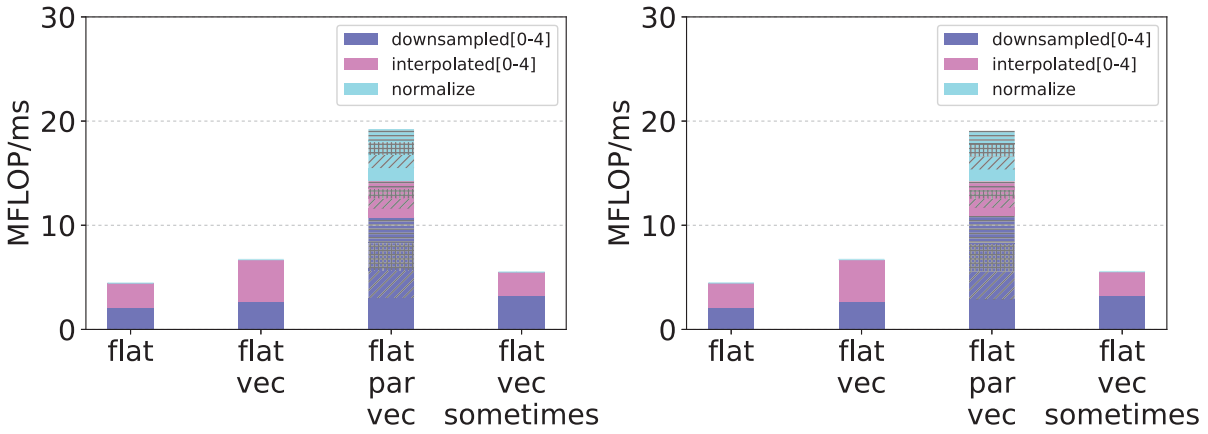


(a) 8 Megapixels (b) 44 Megapixels

Figure 6.13: Number of float instructions for each Interpolate schedule on 8 Megapixel and 44 Megapixel images.

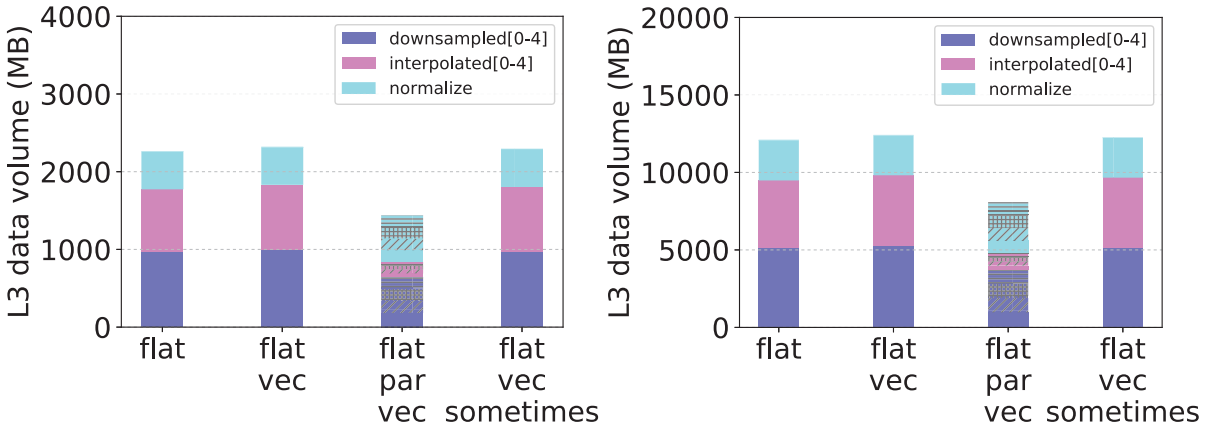
memory but only one element is used. When unrolling the loop, we then use another element from the loaded cache line, therefore reducing the number of misses.

Figures 6.13(a) and 6.13(b) show the number of floating point instructions per schedule for 8 Megapixel and 44 Megapixel image sizes, respectively. In schedules with vectorization,



(a) 8 Megapixels (b) 44 Megapixels

Figure 6.14: Number of float operations per second for each Interpolate schedule on 8 Megapixel and 44 Megapixel images.



(a) 8 Megapixels (b) 44 Megapixels

Figure 6.15: L3 data volume (in MB) for each Interpolate schedule on 8 Megapixel and 44 Megapixel images.

each vector instruction performs four operations, which explains why there is a higher number of instructions in the **flat** schedule compared to the others (each instruction in this schedule just performs one operation), and the **flat_vec_sometimes** has the second highest number of instructions.

Figures 6.14(a) and 6.14(b) show the number of floating point instructions per milisecond for each schedule, for 8 Megapixel and 44 Megapixel image sizes, respectively. Notice that in this case, the schedules with vectorization performs much more operations per milliseconds, which shows very good use of vectorization. A good way to analyze this in the illustrated charts is to compare the **normalize** stage in the **flat_par_vec** schedule with the other schedules. Since **flat_par_vec** is the only schedule that performs vectorization during the **normalize** stage, the CPU throughput ends up being significantly higher in it than in the other schedules.

Finally, Figures 6.15(a) and 6.15(b) show the L3 data volume per schedule for 8 Megapixel and 44 Megapixel image sizes, respectively. In this algorithm we have a more successful case of parallelism than in the Blur algorithm. Notice that the data volume in the parallel schedule is lower than the other schedules. This is a consequence of both the smaller number of cache misses because of the unrolling, and the efficient parallelism strategy that avoids redundant memory accesses among different threads.

6.3 DISCUSSION

6.3.1 FLOP Overcounting

Even thought we used the FLOP counters during our experiments, they are not reliable to accurately measure the number of flops. This happens because in some processors (more specifically Intel Sandy Bridge and Ivy Bridge processors), the counter is incremented when the instructions are issued, and not executed or retired. Eventually, instructions can be rejected if their inputs are not ready yet, which turns them to be rejected after issued, and then re-issued another time. Such FLOP over-counting tends to be higher in applications with bad locality, because the latency time is higher for the inputs to be ready.

Because of this difficulty, it is misleading to suggest that this counter can be used to monitor extra / redundant computation overhead. A better way to measure how many redundant computations have been introduced is to analyze the Halide generated code directly. This is already present in Halide [45] and could be integrated to our profiler extension.

Notice that this example of FLOP overcounting is important because it shows an anomaly known for certain events (flops counter) in some particular processors. Although this just happens in a very small fraction of the cases, it enhances the importance of understanding the concepts, limitations and eventual pitfalls of the events being counted on specific processors.

6.3.2 Profiling Time

An important aspect when profiling applications is the time costs for executing the instrumented code and count the events. In our experiments, we noticed that there are two major factors that can considerably increase profile time: profiling inner loops and parallel regions.

When profiling the generated codes, we may want to separately measure events on regions that interleaves the production and consumption between different stages. As an example, in the *Blur* algorithm with the *tile 32x32* schedule we want to profile both the production and consumption of **blur_x** values separately. For this, we have to introduce markers in the loops that traverse each tile region. This means that we must call the profiler API at every tile (which in this case is at each 32x32 block of pixels). Such excessive calls to the profiler API severely increases the profiling time.

For parallel regions, we have to create a different PAPI event set per thread. This requires the creation and initialization of the event set when the thread arises, and its destruction when it vanishes. Such management of event sets for each thread can also considerable influence the execution time of the profiled code.

schedule	**without profiler**		**with profiler**		**relative**
	avg (ms)	**stdev**	**avg (ms)**	**stdev**	**incr. (%)**
serial breadth-first	22.61	0.42	22.81	0.08	0.8
serial full-fusion	10.84	0.03	11.92	0.05	9.96
serial sliding-window	36.46	0.53	36.94	0.07	1.31
serial tile 32x32	13.02	0.04	226.38	0.90	1638.70
parallel breadth-first	14.94	1.11	1485.71	27.44	9844.51
parallel full-fusion	3.45	1.08	732.12	10.08	21120.86
parallel tile 32x32	5.37	1.30	237.06	2.37	4314.52

Table 6.1: Profiling time results for each Blur schedule on 8 Megapixel images.

schedule	without profiler		with profiler		relative
	avg (ms)	stdev	avg (ms)	stdev	incr. (%)
serial breadth-first	120.00	0.39	120.12	0.38	0.1
serial full-fusion	63.08	0.15	63.08	0.15	0.0
serial sliding-window	196.94	0.44	198.49	0.56	0.78
serial tile 32x32	75.13	0.07	1196.81	4.63	1492.98
parallel breadth-first	85.92	1.19	3348.74	32.56	3797.50
parallel full-fusion	30.30	1.08	1672.91	25.94	5421.15
parallel tile 32x32	42.63	1.31	1209.06	6.42	2736.17

Table 6.2: Profiling time results for each Blur schedule on 44 Megapixel images.

schedule	without profiler		with profiler		relative
	avg (ms)	stdev	avg (ms)	stdev	incr. (%)
flat	256.43	0.51	256.56	0.44	0.05
flat par vec	56.53	0.27	744.10	13.20	1216.29
flat vec sometimes	216.88	0.43	217.06	0.50	0.08
flat vec	193.90	0.43	194.26	0.45	0.18

Table 6.3: Profiling time results for each Interpolate schedule on 8 Megapixel images.

schedule	without profiler		with profiler		relative
	avg (ms)	stdev	avg (ms)	stdev	incr. (%)
flat	1357.56	1.29	1357.49	1.36	0.0
flat par vec	306.68	0.61	2234.17	38.22	628.50
flat vec sometimes	1146.84	1.67	1147.80	1.30	0.08
flat vec	1027.77	1.67	1028.89	1.61	0.10

Table 6.4: Profiling time results for each Interpolate schedule on 44 Megapixel images.

Tables 6.1 and 6.2 show the time average in milliseconds, their standard deviation and the relative time increase for each schedule in the Blur algorithm on 8 and 44 Megapixel images. The tables show the results for both versions (with and without the profiler markers). The standard deviation in all cases is negligible compared to the average, except for the parallel *full-fusion* and *tile-32x32* schedules on 8 Megapixel images. This happens because the time for these schedules is very small (less than 6ms), and a simple time variation in different iterations caused by external factors (such as operating system and hardware) is considerable, increasing the standard deviation.

We can observe by the presented results that the performance penalty on parallel schedules is severe for 8 Megapixel images (the instrumented code is around 212× slower in the parallel *full-fusion* schedule). The proportion is smaller for 44 Megapixel images (in the same parallel *full-fusion* schedule, the instrumented code is about 55× slower). This indicates that a

considerable fraction of the performance penalty in these examples is a overhead time probably occasioned when internally managing the PAPI event sets. Larger image sizes are necessary in order to have a conclusion on how much profiling parallel regions impacts the performance in this algorithm.

Looking at the serial *tile-32x32* schedule, we can perceive the difference when profiling more internal loops in the loop nest. For 8 Megapixel images, the relative time increase is 1638% (the performance penalty turns the code to execute around 17× slower), whereas for 44 Megapixel images, it is 1492% (the instrumented code executes about 15× slower). In this case, the difference when increasing the image size is smaller, which indicates that the performance penalty for this case using bigger image sizes should not be too different of this proportion.

Tables 6.3 and 6.4 show the time average in milliseconds, their standard deviation and the relative time increase for each schedule in the Interpolate algorithm on 8 and 44 Megapixel images. Standard deviation in all cases is negligible compared to the average. For almost all the schedules, there is no performance penalty since they are flat schedules and no profiling of loops in inner loops of the nest was necessary. The only schedule that suffered a considerable increase in time was the *flat_par_vec* schedule due to parallelism. Nevertheless, the behavior is not very different than the presented in the blur parallel schedules. The relative time increase is 1216% for 8 Megapixel images when profiling (the instrumented code executes around 13× slower), and 628% for 44 Megapixel images (the instrumented code executes about 7× slower).

As an early attempt to get better performing profiling codes, we introduced the **granularity** parameter in order to just profile some iterations of the code, which corresponds to perform sampling instead of accurate counting of the events. This helps when dealing with the more internal loops issue, which is the most critic one. For parallel regions profiling, there is no much thing we could do to enhance the performance of the profiling code.

Profiling time is more relevant when we are dealing with automatic scheduling of Halide applications. Because when a developer is manually tuning the application, it is debugging and analyzing different schedules through successive executions, and therefore the execution time is not too significant (except if the profiling time penalty turns the code impracticable to wait for, which is not the case).

Even considering that some schedules may take longer time to run when having the profiler annotations, the information obtained still can compensate this when using an auto-scheduler since it will lead the auto-scheduler to converge much faster in comparison to using less information. Also, some strategies that use online profiling data interrupts executions that reach certain time limits [16, 73], in our case this auto-schedulers could interrupt the execution and increase the granularity until the limit is not reached.

Finally, it is still possible to decrease such time by using a different profiling tool. Since we presented a modular design, it is possible to use a different online profiling tool that could cause less time overhead for the instrumented code.

6.3.3 GPU Results

We use this subsection to report the problems we have on profiling GPU schedules. We were able to insert the appropriate instrumentation markers for profiling GPU kernels, but all counters were set to zero after the execution of the instrumented code.

After investigating, we noticed that Halide CUDA runtime code uses the driver API library [50] to prepare and launch the kernels. At some point, the CUDA runtime code creates a separated context to execute the kernel with the **cuCtxCreate** function. Performing some experiments, we concluded that the PAPI CUDA module cannot profile kernels executing on a

different context, and therefore we cannot perform the experiments for CUDA GPU code at the moment.

To overcome this issue, we could change the Halide runtime code to do not run kernels in a separate context — which could lead to different behavior of the instrumented code in relation to the production code. Another possibility would be to either replace the PAPI library with another tool that can profile CUDA kernels on different context, or to improve the PAPI library so it can profile CUDA kernels running on a different context.

7 CONCLUSIONS AND FUTURE WORK

In this work we presented a Halide DSL extension for profiling Halide code through instrumentation of the generated code with performance events counting functions. Profiling hardware events can help developers and auto-tuners, *i.e.*, to check whether specific Halide functions in the pipeline must be scheduled to optimize memory accesses or to avoid redundant computation. In deeper and more complex pipelines, finding the most appropriate optimization for each stage becomes much more challenging without the appropriate feedback.

The proposed Halide profiling extension offers three simple but powerful functions that allow the obtainment of important performance data from different regions of a Halide pipeline. To demonstrate the use of the proposed extension we profiled the *Blur* algorithm with four different schedules. We used the PAPI library for exposing CPU events and measured time, L1 cache misses, FLOP, and L3 data volume on four different versions of each schedule: serial, serial with vectorization, parallel and parallel with vectorization. We also profiled the *Interpolation* algorithm with four different schedules to show an algorithm with a different behavior.

We discussed some aspects regarding our results. More specifically, we talk about the FLOP overcounting problem and how the FLOP counter used can mislead the counting of redundant computation. Another importat thing that we put in discussion is the increase of the execution time when using the instrumented code, we focus our discussion in the two factors that most increase the profiler time, which are the profiling of more internal loops and parallel regions. And we end our discussion reporting the issues we have when profiling Halide generated code for CUDA GPUs.

Our experimental results show that the extension provides very important performance information to Halide developers, making it easier to spot performance issues on their schedulers, considering the "right questions are asked". We demonstrate that the number of cache misses in the *breadth first* vastly increases due to its poor data locality, and the number of FLOP in the *full fusion* schedule is also higher than the other schedules since it does not store intermediate values and performs more redundant computations. We also show that the parallel versions of *breadth first* and *full fusion* schedules redundant memory access. Additionaly, we show that the parallelism strategy and vectorization for the *Interpolation* algorithm performs better as it is a computation-intensive algorithm and the parallel strategy does not schedule redundant memory access. The effects of the loop unrolling in the *flat_par_vec* schedule also shows a reduced number of cache misses in the interpolation stages, which can be explained due to better utilization of the cache lines.

Our approach is very flexible by allowing the integration of different profiling tools. Even though we have used PAPI in this paper, it is possible to change the profiling library by another that best suits the user requirements or hardware platforms not supported by PAPI. The choice of events to be profiled is not limited by our proposal, and the choice of events should be directed by the optimization target being pursued, which can be runtime, or runtime and energy consumption.

7.1 FUTURE WORK

The profiler's output can be used for an auto-tuner to automatically generate better schedules. As future work, we intend to integrate our profiler to a machine learning based auto-scheduler in

order to achieve a faster convergence on optimal schedules by feeding the auto-scheduler with more sensitive performance data.

In order to integrate the profiler to an auto-scheduler, we would have to automatically determine the stages for performance measurement. For now, we expect the developer to decide at which regions the markers must be inserted. An initial idea would be to profile all the functions that are computed at root levels. Using, for instance, the Halide auto-scheduling work that performs grouping of Halide functions [45], this approach would profile each group separately. In successive measurements, the auto-scheduler could then advanced to profile inner loops for more fine-grained measurements.

Also, as mentioned, our approach allows to use different profiling tools besides PAPI. We also intend to use different tools and compare the counted events and time overhead of the instrumented code for each one.

Another possibility for the automatic generation of better schedules would be to use a mathematical model to define the schedules. The schedules could be then built based on the target architecture features (*i.e.* cache sizes and processor speed) and the algorithm stages features (*i.e.* number of operations and memory access pattern). In this approach, the profiler extension is useful to validate the model through its counted events.

Finally, we could allow our profiler extension to export the results to a standard graphic format, and to display error or warning messages in case of profiling unavailable regions in the loop nest. This would lead to a better user experience for the schedule developers when analyzing the obtained results, specially when dealing with very large pipelines.

REFERENCES

[1] Norman Adams, David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, and James Philbin. Orbit: An optimizing compiler for scheme. In *Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, pages 219–233, Palo Alto, California, USA, June 1986.

[2] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, Cambridge, 1992.

[3] Innovative Computing Laboratory at the University of Tennessee. Papi 3.5 user guide. `http://icl.cs.utk.edu/projects/papi/files/documentation/PAPI_USER_GUIDE.htm`, 2018. Accessed in 06/08/2018.

[4] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of openmp tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, March 2009.

[5] Oleh Berezsky, Oleh Pitsun, Lesia Dubchak, Petro Liashchynskyi, and Pavlo Liashchynskyi. Gpu-based biomedical image processing. In *2018 XIV-th International Conference on Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, pages 96–99, Lviv, Ukraine, Ukraine, April 2018.

[6] Valentin Biaud, Vincent Despiegel, Catherine Herold, Olivier Beiler, and Stéphane Gentric. Semi-supervised evaluation of face recognition in videos. In *Proceedings of the International Workshop on Video and Image Ground Truth in Computer Vision Applications*, VIGTA '13, pages 1:1–1:6, New York, NY, USA, 2013. ACM.

[7] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[8] Quan Chen and Minyi Guo. Locality-aware work stealing based on online profiling and auto-tuning for multisocket multicore architectures. *ACM Trans. Archit. Code Optim.*, 12(2):22:1–22:24, July 2015.

[9] Quan Chen, Minyi Guo, and Haibing Guan. Laws: Locality-aware work-stealing for multi-socket multi-core architectures. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 3–12, New York, NY, USA, 2014. ACM.

[10] Quan Chen, Long Zheng, and Minyi Guo. Adaptive demand-aware work-stealing in multi-programmed multi-core architectures. *Concurr. Comput. : Pract. Exper.*, 28(2):455–471, February 2016.

[11] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Patus : A code generation and auto-tuning framework for parallel iterative stencil computations on modern microarchitectures. In *IEEE International Parallel & Distributed Processing Symposium*, 2011.

[12] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Vol. 3*. Intel Corporation.

[13] Terpstra D., H. Jagode, H. You, and J. Dongarra. Collecting performance data with papi-c. In *3rd Parallel Tools Workshop*, pages 157–173, 2009.

[14] Halide developers. Halide. `http://halide-lang.org/`, 2018. Accessed in 06/09/2018.

[15] Linux Kernel developers. Perf wiki. `https://perf.wiki.kernel.org/index.php/Main_Page`, 2019. Accessed in 06/06/2019.

[16] Yuri Dotsenko, Sara S. Baghsorkhi, Brandon Lloyd, and Naga K. Govindaraju. Auto-tuning of fast fourier transform on graphics processors. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 257–266, New York, NY, USA, 2011. ACM.

[17] Adriana Draghici and Maarten Van Steen. A survey of techniques for automatically sensing the behavior of a crowd. *ACM Comput. Surv.*, 51(1):21:1–21:40, February 2018.

[18] Anders Eklund, Paul Dufort, Daniel Forsberg, and Stephen M. LaConte. Medical image processing on the gpu - past, present and future. *Medical Image Analysis*, 17:1073–1094, December 2013.

[19] ExaStencils. Exastencils. `http://www.exastencils.org/de/`, 2018. Accessed in 05/14/2018.

[20] Apostolos Gerasoulis and Tao Yang. A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors. *Journal of Parallel and Distributed Computing*, 16(4):276 – 291, 1992.

[21] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 39(4):49–57, April 2004.

[22] Martin Griebl, Christian Lengauer, and Sabine Wetzel. Code generation in the polytope model. In *In IEEE PACT*, pages 106–111. IEEE Computer Society Press, 1998.

[23] João Guerreiro, Aleksandar Ilic, Nuno Roma, and Pedro Tomás. Multi-kernel auto-tuning on gpus: Performance and energy-aware optimization. In *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2015.

[24] Chris Harris and Mike Stephens. A combined corner and edge detector. In *Fourth Alvey Vision Conference*, pages 147–151, 1988.

[25] Hendry and Rung-Ching Chen. A new method for license plate character detection and recognition. In *Proceedings of the 6th International Conference on Information Technology: IoT and Smart City*, ICIT 2018, pages 204–208, New York, NY, USA, 2018. ACM.

[26] Justin Holewinski, Louis-Noël Pouchet, and P Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *International conference on Supercomputing*, pages 311–320, 2012.

[27] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv*, 2017.

[28] Graham Hutton and Erik Meijer. Monadic parser combinators. Technical report, Departament of Computer Science, University of Notthingham, 1996.

[29] Anastasia Izmaylova, Ali Afroozeh, and Tijs van der Storm. Practical, general parser combinators. In *Symposium on Partial Evaluation and Program Manipulation (PEPM)*, 2016.

[30] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *ACM SIGPLAN conference on Programming Languages Design and Implementation*, 2007.

[31] Marcel Köster, Roland Leißa, Sebastian Hack, Richard Membarth, and Philipp Slusallek. Code refinement of stencil codes. *Parallel Processing Letters 2014*, 24, 2014.

[32] Marcel Köster, Roland Leißa, Sebastian Hack, Richard Membarth, and Philipp Slusallek. Platform-specific optimization and mapping of stencil codes through refinement. In *Proceedings of the 1st International Workshop on High-Performance Stencil Computations (HiStencils)*, 2014.

[33] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[34] Roland Leißa, Klaas Boesche, Sebastian Hack, Richard Membarth, and Philipp Slusallek. Shallow embedding of dsls via online partial evaluation. In *Proceedings of the 14th International Conference on Generative Programming: Concepts & Experiences (GPCE)*, pages 11–20, Pittsburgh, PA, USA, October 2015.

[35] Roland Leißa, Marcel Köster, and Sebastian Hack. A graph-based higher-order intermediate representation. In *Proceedings of the 2015 International Symposium on Code Generation and Optimization (CGO)*, pages 202–212, San Francisco, CA, USA, February 2015.

[36] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. Differentiable programming for image processing and deep learning in Halide. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 37(4):139:1–139:13, 2018.

[37] Rafael Ravedutti Lucio Machado, Andre Murbach Maidl, and Daniel Weingaertner. Profiling halide DSL with CPU performance events for schedule optimization. In *XXIII Brazilian Symposium on Programming Languages, SBLP 2019*, 2019.

[38] Richard Membarth and Oliver Reiche. Hipacc. `https://hipacc-lang.org/`, 2018. Accessed in 06/09/2018.

[39] Richard Membarth, Oliver Reiche, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert. Hipacc, a domain-specific language and compiler for image processing. In *Transactions on Parallel and Distributed Systems (TPDS)*, pages 210–224, 2016.

[40] Richard Membarth, Philipp Slusallek, Marcel Köster, Roland Leißa, and Sebastian Hack. Target-specific refinement of multigrid codes. In *Proceedings of the 4th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, pages 52–57, New Orleans, LA, USA, November 2014.

[41] Jiayuan Meng and Kevin Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 256–265, New York, NY, USA, 2009. ACM.

[42] Antoine Monsifrot, François Bodin, and Rene Quiniou. A machine learning approach to automatic production of compiler heuristics. In *Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, AIMSA '02, pages 41–50, London, UK, UK, 2002. Springer-Verlag.

[43] Ramon E. Moore. *Interval Analysis*. Society for Industrial and Applied Mathematics, 1967.

[44] MPI. Mpi: A message-passing interface standard. Technical report, University of Tennessee Knoxville, TN, USA, 1994.

[45] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *Proceedings of ACM SIGGRAPH*, July 2016.

[46] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. Polymage: Automatic optimization for image processing pipelines. In *Proceedings of the Twentieth International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 429–443, Istanbul, Turkey, March 2015.

[47] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, and Koen Bertels. A survey and evaluation of fpga high-level synthesis tools. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1591–1604, December 2016.

[48] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[49] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *ueue - GPU Computing*, 6:40–53, March/April 2008.

[50] NVIDIA. Cuda driver api on cuda toolkit documentation. `https://docs.nvidia.com/cuda/cuda-driver-api/index.html`, 2019. Accessed in 10/07/2019.

[51] OpenMP. Specifications - openmp. `http://www.openmp.org/specifications/`, 2018. Accessed in 05/14/2018.

[52] Chetan Patil, Mathura M G, Madhumitha S, Sumam David S, Merwyn Fernandes, Anand Venugopal, and B. Unnikrishnan. Early detection of alzheimer's disease - using image processing on mri scans. In *IEEE International Conference on Signal Processing, Informatics, Communication and Energy Systems (SPICES)*, February 2015.

[53] Marcelo Pecenin. Otimização do escalonamento halide através de aprendizado por reforço. Master's thesis, Pós-Graduação em Informática - Universidade Federal do Paraná, Curitiba - PR, February 2019.

[54] Arsène Pérard-Gayot, Martin Weier, Richard Membarth, Philipp Slusallek, Roland Leißa, and Sebastian Hack. Ratrace: Simple and efficient abstractions for bvh ray traversal algorithms. In *Proceedings of the 16th International Conference on Generative Programming: Concepts & Experiences (GPCE)*, pages 157–168, Vancouver, BC, Canada, October 2017.

[55] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo. Spiral: Code generation for dsp transforms. In *Proceedings of the IEEE*, pages 232–275, Istanbul, Turkey, February 2005.

[56] Jonathan Ragan-Kelley. *Decoupling Algorithms from the Organization of Computation for High Performance Image Processing: The design and implementation of the Halide language and compiler*. PhD thesis, MIT - Massachusetts Institute of Technology, Massachusetts - United States, May 2014.

[57] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, and Saman Amarasinghe. Decoupling algorithms from schedules for easy optimization of image processing pipelines. In *ACM Transactions on Graphics 31(4) (In Proceedings of SIGGRAPH 2012)*, 2012.

[58] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *PLDI 2013*, 13, June 2013.

[59] Oliver Reiche, Richard Membarth, Jürgen Teich, and Frank Hannig. Code generation for embedded heterogeneous architectures on android. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, March 2014.

[60] Oliver Reiche, Mehmet Akif Özkan, Richard Membarth, Jürgen Teich, and Frank Hannig. Generating fpga-based image processing accelerators with hipacc. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pages 13–16, Irvine, CA, USA, November 2017.

[61] N. Garner G. Ho S. Browne, J. Dongarra and P. Mucc. A portable programming interface for performance evaluation on modern processors. In *International Journal of High Performance Computing Applications*, pages 189–204, 2000.

[62] Christian Schmitt, Sebastian Kuckuk, Frank Hannig, Harald Köstler, and Jürgen Teich. Exaslang: A domain-specific language for highly scalable multigrid solvers. In *Proceedings of the 4th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, pages 42–51, Izmir, Turkey, November 2014.

[63] Tarundeep Singh, Nawwaf Kharma, Mohmmad Daoud, and Rabab Ward. Genetic programming based image segmentation with applications to biomedical object detection. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 1123–1130, New York, NY, USA, 2009. ACM.

[64] Robert Stewart. An image processing language: External and shallow/deep embeddings. In *Proceedings of the 1st International Workshop on Real World Domain Specific Languages*, RWDSL '16, pages 6:1–6:10, New York, NY, USA, 2016. ACM.

[65] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Design & Test*, 12:66–73, May 2010.

[66] Jan Treibig, Georg Hager, and Georg Hager. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *ICPPW '10 Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, pages 207–216, 2010.

[67] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In *International Congress Conference on Mathematical Software*, pages 299–302, 2010.

[68] Ritika Verma and Indu Sreedevi. Robust pedestrian tracking using improved tracking-learning-detection algorithm. In *Proceedings of the Tenth Indian Conference on Computer Vision, Graphics and Image Processing*, ICVGIP '16, pages 35:1–35:8, New York, NY, USA, 2016. ACM.

[69] Z. Wang and M. O'Boyle. Machine learning in compiler optimization. *Proceedings of the IEEE*, 106(11):1879–1901, Nov 2018.

[70] Vincent M. Weaver, Matt Johnson, Kiran Kasichayanula, James Ralph, Piotr Luszczek, Dan Terpstra, and Shirley Moore. Measuring energy and power with papi. In *Proceedings of the 2012 41st International Conference on Parallel Processing Workshops*, pages 262–268, 2012.

[71] Nicolas Weber, Sandra C. Amend, and Michael Goesele. Guided profiling for auto-tuning array layouts on gpus. In *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems*, 2015.

[72] Nicolas Weber and Michael Goesele. Adaptive gpu array layout auto-tuning. In *Software Engineering Methods for Parallel and High Performance Applications*, 2016.

[73] Nicolas Weber and Michael Goesele. Matog: Array layout auto-tuning for cuda. *ACM Trans. Archit. Code Optim.*, 14(3):28:1–28:26, August 2017.

[74] Peng Zhang, Tony Thomas, Sabu Emmanuel, and Mohan S. Kankanhalli. Privacy preserving video surveillance using pedestrian tracking mechanism. In *Proceedings of the 2nd ACM Workshop on Multimedia in Forensics, Security and Intelligence*, MiFor '10, pages 31–36, New York, NY, USA, 2010. ACM.

[75] W. Zhao, R. Chellappa, P. J. Phillips, and A. Rosenfeld. Face recognition: A literature survey. *ACM Comput. Surv.*, 35(4):399–458, December 2003.

[76] Xing Zhou, Jean-Pierre Giacalone, María Jesús Garzarán, Robert H. Kuhn, Yang Ni, and David Padua. Hierarchical overlapped tiling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 207–218, New York, NY, USA, 2012. ACM.

APPENDIX A – BLUR CODE USED FOR EXPERIMENTS

Here we show the Halide codes used in this work for both the *Blur* application algorithm and different schedules.

A.1 BLUR ALGORITHM

```
Buffer<float> input(3840, 2160, 1);
//Buffer<float> input(10240, 4320, 1);
Func blur_x, blur_y;
Var x, y, c, xi, yi;

blur_x(x, y, c) = (
  input(x - 1, y, c) + input(x, y, c) + input(x + 1, y, c)
) / 3.0f;
blur_y(x, y, c) = (
  blur_x(x, y - 1, c) + blur_x(x, y, c) + blur_x(x, y + 1, c)
) / 3.0f;
```

A.2 BLUR SCHEDULES

```
#if SCHEDULE == 1
   /* Breadth-first */
   schedule = "breadth_first";

   blur_x.compute_root();

  #ifdef VECTORIZE
   blur_x.vectorize(x, 4);
   blur_y.vectorize(x, 4);
  #endif

  #ifdef PARALLEL
   blur_x.parallel(y);
   blur_y.parallel(y);
  #endif

  #ifdef PROFILE
   blur_x.profile(PROFILE_PRODUCTION, true, true);
   blur_y.profile(PROFILE_PRODUCTION, true, true);
  #endif

#elif SCHEDULE == 2
   /* Full-fusion */
   schedule = "full_fusion";

  #ifdef VECTORIZE
   blur_y.vectorize(x, 4);
  #endif

  #ifdef PARALLEL
   blur_y.parallel(y);
```

```
32    #endif
33
34    #ifdef PROFILE
35      blur_x.profile(PROFILE_PRODUCTION, true, true);
36      blur_y.profile(PROFILE_PRODUCTION, true, true);
37    #endif
38
39  #elif SCHEDULE == 3
40      /* Sliding window */
41      schedule = "sliding_window";
42
43      blur_x.store_at(blur_y, c).compute_at(blur_y, x);
44
45    #ifdef VECTORIZE
46      blur_x.vectorize(x, 4);
47      blur_y.vectorize(x, 4);
48    #endif
49
50    #ifdef PARALLEL
51      blur_y.parallel(y);
52    #endif
53
54    #ifdef PROFILE
55      profile_at(blur_y, c, true);
56    #endif
57
58  #elif SCHEDULE == 4
59      /* Tile (block dimension = 32x32) */
60      schedule = "tile_32x32";
61
62      blur_y.tile(x, y, xi, yi, 32, 32);
63      blur_x.compute_at(blur_y, x);
64
65    #ifdef VECTORIZE
66      blur_y.vectorize(xi, 4);
67      blur_x.vectorize(x, 4);
68    #endif
69
70    #ifdef PARALLEL
71      blur_y.parallel(y);
72    #endif
73
74    #ifdef PROFILE
75      blur_x.profile(PROFILE_PRODUCTION, true, true);
76      blur_y.profile(PROFILE_PRODUCTION, true, true);
77    #endif
78
79  #else
80      schedule = "invalid";
81      #error "Invalid schedule!"
82  #endif
83
84    return 0;
85  }
```

APPENDIX B – INTERPOLATION CODE USED FOR EXPERIMENTS

Here we show the Halide codes used in this work for both the *Interpolation* application algorithm and different schedules.

B.1 INTERPOLATION ALGORITHM

```
ImageParam input(Float(32), 3);

// Input must have four color channels - rgba
input.dim(2).set_bounds(0, 4);

const int levels = 5;

Func downsampled[levels];
Func downx[levels];
Func interpolated[levels];
Func upsampled[levels];
Func upsampledx[levels];
Var x("x"), y("y"), c("c");

Func clamped = BoundaryConditions::repeat_edge(input);

downsampled[0](x, y, c) = clamped(x, y, c) * clamped(x, y, 3);

for (int l = 1; l < levels; ++l) {
   Func prev = downsampled[l-1];

   if (l == 4) {
      Expr w = input.width()/(1 << l);
      Expr h = input.height()/(1 << l);
      prev = lambda(x, y, c, prev(clamp(x, 0, w), clamp(y, 0, h), c));
   }

   downx[l](x, y, c) = (prev(x*2-1, y, c) +
                        2.0f * prev(x*2, y, c) +
                        prev(x*2+1, y, c)) * 0.25f;
   downsampled[l](x, y, c) = (downx[l](x, y*2-1, c) +
                              2.0f * downx[l](x, y*2, c) +
                              downx[l](x, y*2+1, c)) * 0.25f;
}
interpolated[levels-1](x, y, c) = downsampled[levels-1](x, y, c);
for (int l = levels-2; l >= 0; --l) {
   upsampledx[l](x, y, c) = (interpolated[l+1](x/2, y, c) +
                             interpolated[l+1]((x+1)/2, y, c)) / 2.0f;
   upsampled[l](x, y, c) = (upsampledx[l](x, y/2, c) +
                            upsampledx[l](x, (y+1)/2, c)) / 2.0f;
   interpolated[l](x, y, c) = downsampled[l](x, y, c) +
                              (1.0f - downsampled[l](x, y, 3)) *
                              upsampled[l](x, y, c);
}

Func normalize("normalize");
normalize(x, y, c) = interpolated[0](x, y, c) / interpolated[0](x, y, 3);
```

B.2 INTERPOLATION SCHEDULES

```
#if SCHEDULE == 1
   schedule = "flat";

   for (int l = 0; l < levels; ++l) {
     downsampled[l].compute_root();
     interpolated[l].compute_root();
   }

   normalize.compute_root();

#elif SCHEDULE == 2
   schedule = "flat_vec";

   for (int l = 0; l < levels; ++l) {
     downsampled[l].compute_root().vectorize(x,4);
     interpolated[l].compute_root().vectorize(x,4);
   }

   normalize.compute_root();

#elif SCHEDULE == 3
   schedule = "flat_par_vec";

   Var xi, yi;

   for (int l = 1; l < levels-1; ++l) {
     downsampled[l]
       .compute_root()
       .parallel(y, 8)
       .vectorize(x, 4);

     interpolated[l]
       .compute_root()
       .parallel(y, 8)
       .unroll(x, 2)
       .unroll(y, 2)
       .vectorize(x, 8);
   }

   normalize
     .reorder(c, x, y)
     .bound(c, 0, 3)
     .unroll(c)
     .tile(x, y, xi, yi, 2, 2)
     .unroll(xi)
     .unroll(yi)
     .parallel(y, 8)
     .vectorize(x, 8)
     .bound(x, 0, input.width())
     .bound(y, 0, input.height());

#elif SCHEDULE == 4
   schedule = "flat_vec_sometimes";

   for (int l = 0; l < levels; ++l) {
     if (l + 4 < levels) {
```

```
57        Var yo,yi;
58        downsampled[l].compute_root().vectorize(x,4);
59        interpolated[l].compute_root().vectorize(x,4);
60       } else {
61        downsampled[l].compute_root();
62        interpolated[l].compute_root();
63       }
64     }
65
66     normalize.compute_root();
67
68  #else
69     #error "Invalid schedule!"
70  #endif
71
72  #ifdef PROFILE
73
74     for(int l = 0; l < levels; ++l) {
75       downsampled[l].profile(PROFILE_PRODUCTION, true, true);
76       interpolated[l].profile(PROFILE_PRODUCTION, true, true);
77       normalize.profile(PROFILE_PRODUCTION, true, true);
78     }
79
80  #endif
```